2021

Rv-Enabled Framework For Self-Adaptive Software

Teng Zhang
University of Pennsylvania

Rv-Enabled Framework For Self-Adaptive Software

Abstract

Software systems keep increasing in scale and complexity, requiring ever more effort to design, build, test, and deploy. Systems are integrated from separately developed modules. Over the life of a system, individual modules may be updated, which may allow incompatibilities between modules to slip in. Consequently, many faults in a software system are often discovered after the system is built and deployed.

Runtime verification (RV) is a collection of dynamic techniques for detecting faults of software systems. An executable monitor is constructed from a formally specified property of the system being checked (denoted as the target system) and is run over a stream of observations (events) to check whether the property is satisfied or not. Although existing tools are able to specify and monitor properties efficiently, it is still challenging to apply RV to large-scale real-world applications. From the perspective of monitoring requirements, we need a formalism that can describe both high and low-level behaviors of the target system. Complexity of the target program also brings some issues. For instance, it may contain a set of loosely-coupled components which may be added or removed dynamically. Correspondingly, monitoring requirements are often defined upon asynchronous observations that carry data of which the domain scale up along with expansion of the target system. How to conveniently specify these properties and generate monitors that can check them efficiently is a challenge.

Beyond detecting faults, self-adaptive software is desirable for tolerating faults or unexpected environment changes during execution. By equipping monitors with reflexive adaptation actions, runtime enforcement (RE) can be used to improve robustness of the system. However, there is little work on analyzing possible interference between the implementation of adaptation actions and the target program.

In this thesis, we present SMEDL, a RV framework using a specification language designed for high usability with respect to expressiveness, efficiency and flexible deployment. The property specification is composed of a set of communicating monitors described in the form of EFSMs (extend finite state machines). High-level properties can be straightforwardly transformed into SMEDL specifications while actions can be specified in transitions to express low-level imperative behaviors. Deployment of monitors can be explicitly specified to support both centralized and distributed software. Based on dynamically scalable monitor structure, we propose a novel method to efficiently check parametric properties that rely on the data events carry. To tackle challenges of monitoring timing properties in an asynchronous environment, we propose a conceptual monitor architecture that clearly separates monitoring of time intervals from the rest of property checking.

To support software adaptation, we extend the SMEDL framework to specify enforcement specifications, generate implementations and instrument them into the target system. Analysis of interference between the adaptation implementation and the target system can be performed statically based on Hoare-logic. Instead of building a whole new proof for the target system globally, we present a method to generate local proof obligations for better scalability.

Degree Type

Dissertation

Degree Name

Doctor of Philosophy (PhD)

Graduate Group
Computer and Information Science

First Advisor
Oleg V. Sokolsky

Second Advisor
Insup Lee

Subject Categories
Computer Sciences

RV-ENABLED FRAMEWORK FOR SELF-ADAPTIVE SOFTWARE

Teng Zhang

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2021

Co-Supervisor of Dissertation

Oleg Sokolsky, Research Professor
of Computer and Information Science

Co-Supervisor of Dissertation

Insup Lee, Professor
of Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor of Computer and Information Science

Dissertation Committee

Mayur Naik, Professor of Computer and Information Science

Rajeev Alur, Professor of Computer and Information Science

Stephanie Weirich, Professor of Computer and Information Science

Klaus Havelund, Senior Research Scientist of Jet Propulsion Laboratory

RV-ENABLED FRAMEWORK FOR SELF-ADAPTIVE SOFTWARE

Dedicated this work to my wife Meiyuan, without whom I would never have gotten this far.

ACKNOWLEDGEMENT

I would like to express my most gratitude to my advisors, Prof. Oleg Sokolsky and Prof. Insup Lee. Their invaluable advice, continuous support, and patience have helped me go through all the time of my research and daily life. This dissertation would not be possible without them.

I would like to thank colleagues I was fortunate to work with. I got tremendous help from Peter Gebhard and Dominick Pastore during the development of the SMEDL tool. When I was having a hard time working on Coq, John Wiegley, Theophilos Giannakopoulos and Clément Pit-Claudel gave me a lot of advice on how to go through the steepest part of the learning curve. Gregory Eakman provided real-world case studies, which inspired us to develop new features of SMEDL. We also worked closely with the team from GrammaTech, including Cameron Swords, Zak Fry, Lucja Kot and Jim Inoue, to refactor the API design in the tool and apply SMEDL to more fields such as software adaptation.

Along the way of this journey to the Ph.D. degree, I was also very lucky to meet lifelong friends, including Jyh-Jing, Wei-Hsi, Jizhou, Ang, Chuwei and many more. They have not only helped me a lot in life but also given me some great advice in research.

Finally, I would love to devote my deepest gratefulness to my family and loved ones. In particular, I would like to thank my wife, Meiyuan, who was the first one to encourage me to pursue Ph.D. degree. After we got married, she gave up her career opportunity back home and accompanied me at Penn for 6 years. Without her support, I would have never reached this far.

ABSTRACT

RV-ENABLED FRAMEWORK FOR SELF-ADAPTIVE SOFTWARE

Teng Zhang

Oleg Sokolsky

Insup Lee

Software systems keep increasing in scale and complexity, requiring ever more effort to design, build, test, and deploy. Systems are integrated from separately developed modules. Over the life of a system, individual modules may be updated, which may allow incompatibilities between modules to slip in. Consequently, many faults in a software system are often discovered after the system is built and deployed.

Runtime verification (RV) is a collection of dynamic techniques for detecting faults of software systems. An executable monitor is constructed from a formally specified property of the system being checked (denoted as the target system) and is run over a stream of observations (events) to check whether the property is satisfied or not. Although existing tools are able to specify and monitor properties efficiently, it is still challenging to apply RV to large-scale real-world applications. From the perspective of monitoring requirements, we need a formalism that can describe both high and low-level behaviors of the target system. Complexity of the target program also brings some issues. For instance, it may contain a set of loosely-coupled components which may be added or removed dynamically. Correspondingly, monitoring requirements are often defined upon asynchronous observations that carry data of which the domain scale up along with expansion of the target system. How to conveniently specify these properties and generate monitors that can check them efficiently is a challenge.

Beyond detecting faults, self-adaptive software is desirable for tolerating faults or unexpected environment changes during execution. By equipping monitors with reflexive adaptation actions, runtime enforcement (RE) can be used to improve robustness of the system. However, there is little

work on analyzing possible interference between the implementation of adaptation actions and the target program.

In this thesis, we present SMEDL, a RV framework using a specification language designed for high usability with respect to expressiveness, efficiency and flexible deployment. The property specification is composed of a set of communicating monitors described in the form of EFSMs (extend finite state machines). High-level properties can be straightforwardly transformed into SMEDL specifications while actions can be specified in transitions to express low-level imperative behaviors. Deployment of monitors can be explicitly specified to support both centralized and distributed software. Based on dynamically scalable monitor structure, we propose a novel method to efficiently check parametric properties that rely on the data events carry. To tackle challenges of monitoring timing properties in an asynchronous environment, we propose a conceptual monitor architecture that clearly separates monitoring of time intervals from the rest of property checking.

To support software adaptation, we extend the SMEDL framework to specify enforcement specifications, generate implementations and instrument them into the target system. Analysis of interference between the adaptation implementation and the target system can be performed statically based on Hoare-logic. Instead of building a whole new proof for the target system globally, we present a method to generate local proof obligations for better scalability.

Contents

List of Tables

List of Figures

CHAPTER 1 : Introduction

Modern software systems are increasing in scale and complexity. Systems are integrated from separately developed modules, both vertically and horizontally. Distributed computation are extensively used in the integration of modules. Over the life of a system, modules will be updated, replaced and added to fit the fast-evolving requirements. All these complexities make software systems easier to fail, which may lead to serious consequences.

Testing is widely used to detect defects before deployment. However, it cannot guarantee the software is free of bugs and it is hard to apply testing in a post-deployment manner. In contrast, formal methods use mathematical model to guarantee full correctness of the system. However, formal verification becomes intractable as the system becomes more complicated. Moreover, for systems constructed by components which are treated as black boxes, some problems may happen during runtime because of incompatibility with the environment or malicious attacks. Thus, flaws in a software system are often discovered after the system is built and deployed.

Runtime verification (RV) is a kind of dynamic technique widely used for detecting flaws of software systems. The objective of RV is to check if a run of the system, usually abstracted as a trace of events extracted from an execution, satisfies or violates certain properties. It can be applied in multiple phases of software development. During the development phase, RV can be treated as a testing technique to find bugs by detecting violation of properties. After deployment, it can be used to analyze executions of the target system for diagnosis or generate alerts when the system behaves abnormally.

To further improve the robustness of a system in reaction to internal failures or environment changes dynamically, the concept of self-adaptive software has been brought up. Compared to sophisticated techniques which usually rely on system models or heuristics of existing error handling mechanisms, runtime enforcement (RE) specifies actions in an enforcement monitor (EM) to guarantee satisfaction of properties in event traces. By instrumenting actions back to the system, software adaptation can be achieved. This pattern can be used as sanitizers between two systems or policy

enforcers to guarantee correct use of APIs in the target system. The adaptation is *reflexive* in a sense that instrumented actions are pre-defined and would not permanently solve the problems. However, they can be used as effective temporary repairs and are more feasible for validation.

In this thesis, we aim to propose a RV-enabled software adaptation framework. Figure 1 illustrates the architecture. Monitor specifications are created from properties, which are then compiled into runtime monitors. During runtime, the monitor observes the event trace obtained from the target system and generates verdicts on whether the property is violated. To support software adaptation, program actions can also be generated and integrated into the target system. Applying our framework to real-world software systems need to overcome challenges from multiple aspects.

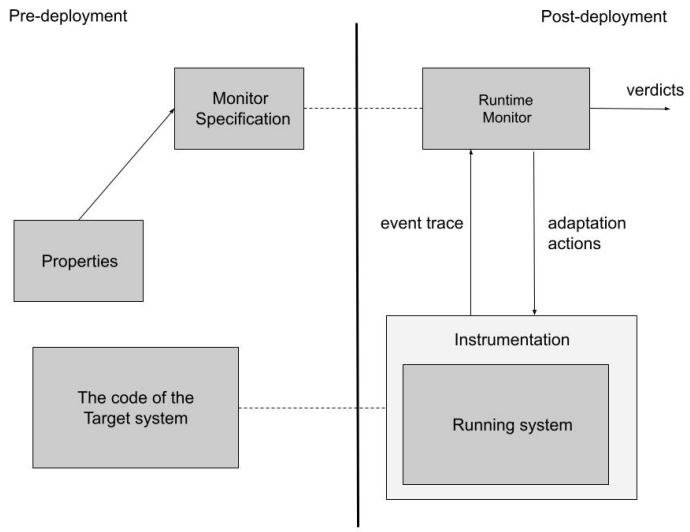

Figure 1: Monitoring/adaptation architecture

1.1. Challenges

1.1.1. Specification language and code generation

First, we want the specification language that can describe a variety of properties for complex software systems and correct and efficient monitor code can be synthesis from the monitor specification. In this thesis, we study the following challenges.

Properties with multiple forms. Many formalisms for runtime verification are suitable to describe

high level properties, such as API policies or protocols. Efficiency of monitoring is guaranteed by well-designed algorithms. For a property involving imperative operations such as arithmetic or logic aggregation, writing a specification in a high level formalism is more challenging, especially for a user with less experience in those formalisms. Instead, using general programming language is more intuitive. However, efficiency depends on the concrete implementation. It is desirable to have a formalism to describe multiple forms of properties in a unified and elegant way.

Properties for complex software. For complex software such as a component-based system, a property may rely on compositional behaviors of multiple components. Using a single monitor to collect observations from different components may incur prohibitive overheads and interfere with system operation. It is more sensible to implement a global property as a set of local monitors of which local verdicts are combined together during execution. As a result, the technique should provide a modular-style specification language and generate monitors that can be deployed in a flexible way.

Correctness of monitor implementations. Executable monitors are generated from property specifications. Informal code generation processes are error-prone and may lead to serious consequences. How to bridge the gap between the specification and the implementation is a challenge.

1.1.2. Monitoring complex properties

Beyond the language and code generation, properties to be monitored also brings challenges. In the thesis, we focus on two types of complicated properties, parametric properties and timing properties.

Monitoring parametric properties. Properties of large-scale systems are often parametric, which means events in a trace may carry data of which the domains expand along with scaling of the system. As an example, we want to guarantee that the iterator of a collection created from a map must not be used after the map has been updated in a program. During execution, maps, collections and iterator objects may be created dynamically. The monitor needs to recognize relationship among them and track their behavior. These properties can be specified and monitored using multiple techniques [75], such as temporal based formalism, stream-processing and trace slicing. These techniques have subtle differences on expressing parametric properties and the monitoring algo-

rithms. For instance, MOP (monitoring oriented programming) [101] and QEA (quantified event automata) [14] both uses the trace slicing method but adopt different slicing semantics. While QEA can express more parametric properties, it is less efficient than MOP [112]. Due to their difference view on parametric properties, it is hard to achieve both expressiveness and efficiency by directly extending either of them. It is thus desirable to have a technique that provides a common ground to describe parametric properties in a unified way and implement parametric monitoring in a flexible fashion, which would not only help people understand distinctions between multiple techniques, but also pave the way towards more efficient monitoring algorithms.

Monitoring timing properties. Timing properties are often defined in the form of checking one event occurring after another event within certain time bound or counting the number of events that occur during an interval of time. In both cases, a monitor needs to not only evaluate the logic of the property but also determine whether events fall within a given time interval. Correctly monitoring interval is challenging in an asynchronous setting where monitors and the target system have different clock and event delivery delays are introduced by unstable network.

1.1.3. Software adaptation

For software adaptation, since enforcement actions may change the state of the program, correctness is a vital issue to handle. Existing work on runtime enforcement have studied principles of specifying enforcement and generate implementations that comply with the specification, there is little work on analyzing how a poor implementation of actions or ill-formed instrumentation may influence execution the target program. We need to define a reasonable correctness criteria, express possible behavior changes after integrating adaptation actions into the target system and statically decide whether the instrumented target system still meets the correctness criteria.

1.2. Contributions

Aiming to solve challenges stated above, we make the following contributions, as shown in Figure 2.

SMEDL: a RV framework. We propose a new formalism, SMEDL (Scenario-based Meta Event

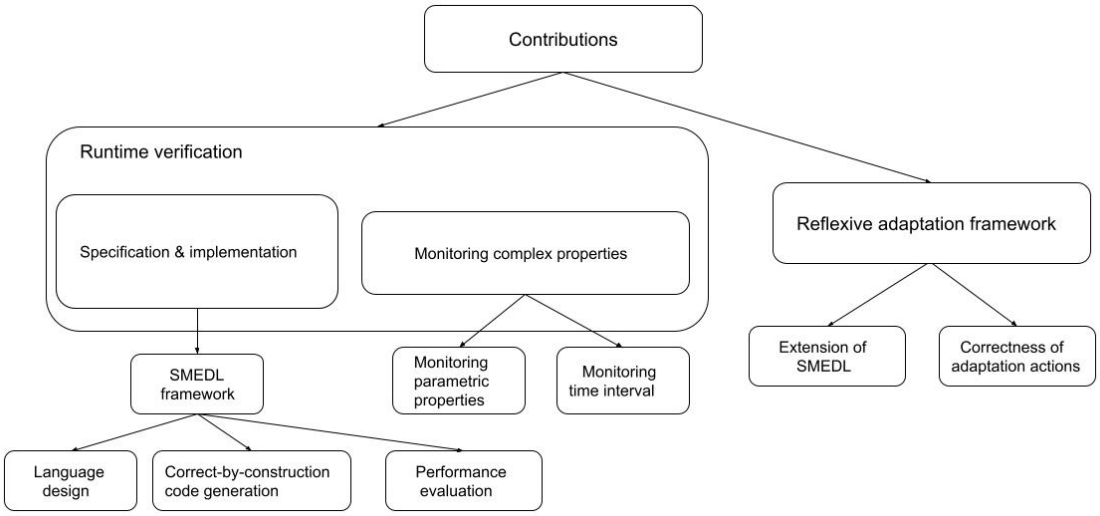

Figure 2: Contributions

Description Language) for property specification [136]. A SMEDL specification is composed of a set of single monitor specifications connecting with each other. Each single monitor is described as composition of EFSMs (extend finite state machines), which is suitable for describing not only high-level temporal properties, but also low-level properties described as explicit state transitions with imperative actions such as logic or arithmetic computations. In the architecture description, users can flexibly specify how monitors are deployed with the target system or communication between monitors, either synchronously or asynchronously. During runtime, monitor instances can be instantiated potentially multiple times to form a dynamically scaling monitor network. The notion of monitor network provides a unified way to compose verdicts of properties and by nature facilitates flexible deployment of monitors to adapt to a variety of software systems such as single process programs and distributed software [138]. To bridge the gap between the specification and the monitor implementation, we formalize the semantics of single monitor execution using Coq[25] to generate the executable code, which is guaranteed to progress in reaction to incoming events and generate deterministic verdicts [139]. To demonstrate usability of our technique, we conduct performance evaluations. We focus on time efficiency of monitor implementations. We compare our technique to representative RV tools in both online and offline settings against multiple benchmark programs and properties. We then profile execution of SMEDL monitors and present intuitions of

optimization from both the perspectives of language design and implementation. Finally, we study how deployment of monitors influence time overheads.

Monitoring parametric properties using SMEDL. By using dynamically scalable monitor network, SMEDL provides a natural and general way to express and monitor parametric properties. To demonstrate this idea, we encode the trace slicing algorithm of MOP by proposing a transformation from MOP to SMEDL. Then, by transforming from QEA specifications into SMEDL monitor network, we can monitor QEA properties more efficiently by applying the trace slicing algorithm of MOP.

Monitoring timing properties in asynchronous environments. We study how to monitor time intervals in an asynchronous environment parameterized by network delay, clock skew and clock rate, and propose a mechanism to separate monitoring of time intervals from the rest of property checking [137].

Reflexive adaptation framework. We extend the SMEDL framework to enable runtime software adaptation. The enforcement specification is expressed as SMEDL monitors with a map from abstract events to concrete adaptation actions. We define the semantic rules to express how adaptation actions may change the behavior of the target program and present a method to generate implementations of actions and integrate them into the program. Based on this implementation, we propose a Hoare-logic based method to verify non-interference of adaptation implementation with respect to the target system. Because proof obligations are generated locally at each instrumentation point, our method can be scaled up to multiple instrumentation points without considering possible inference among actions.

1.3. Structure

The thesis is organized as below. Chapter 2 surveys related work on runtime verification and software adaptation. Chapter 3 presents SMEDL as the specification language. Chapter 4 presents a novel method to specify and monitor parametric properties using SMEDL monitor network. Depending on the knowledge on parametric monitoring, we put the performance evaluation of SMEDL

in Chapter 5. Chapter 6 studies monitoring timing properties in asynchronous environments and proposes a conceptual monitor architecture that clearly separates monitoring of time intervals from the rest of property checking. Chapter 7 extends SMEDL to support software adaptation and presents a Hoare-logic-based method to verify correctness of adaptation implementations with respect to the execution of the target program. Chapter 8 makes conclusions and discusses future work.

CHAPTER 2 : State of the Art

This chapter gives an introduction to the background of the thesis, including a brief account and state of the arts of runtime verification and software adaptation.

2.1. Introduction to runtime verification

Runtime verification (RV), also referred to as runtime monitoring, is a kind of technique for checking properties against behaviors of systems. This section first introduces key concepts of RV and then presents typical applications of runtime verification.

Observations, events, and traces. In RV, behavior of a system is usually represented as a trace of observations obtained from a system execution. Representative types of observations include samples and updates of a system state, calls to functions and transfer of program controls. These observations are abstracted as *events*. The choice of abstraction is determined by the property we care and knowledge and assumptions about the target system to be monitored. For instance, an event can be an atomic proposition on a state variable or a snap shot of the system state. A *trace* is a sequence of events produced by execution of the system, which can be potentially extended indefinitely.

Properties and specification languages. A property is a set of traces. RV techniques usually provide a specification language (also referred to as a formalism) to describe properties. Specification languages can be divided into two styles: declarative and operational style [15]. In declarative-style formalisms, such as temporal logic and its variants, specifications describe what to monitor, which are suitable for high-level behaviors. Operational-style formalisms, such as automata-based languages, describe monitoring logic in an imperative way. It is also possible to transform temporal formula into automata [63].

Monitoring system. Figure 1 captures process of runtime monitoring. Monitor specifications are created from properties, which are then synthesized into runtime monitors. A monitor can be either encoded as a data structure to be interpreted by a monitoring algorithm or compiled into pro-

grams [9]. Monitors can check execution of a system and generate verdicts in an *online* manner when the system is running or in an *offline* manner by consuming an event trace from a log file. In either case, instrumentation is necessary as an interface between the system and the monitor. Not shown in the figure, the instrumentation script specifies how to instrument the target program and map from interesting observations into events to be handled by monitors. Instrumentation techniques then analyze the program, at the source code level using AOP (aspect oriented programming) [84] or using binary analysis, and integrate the code that generates event traces during runtime into the target program.

Application of runtime verification. Here we give two representative examples to illustrate that RV can be applied to verify both high and low-level behavior of a system.

Case 1: Correctness of using library APIs. Many programs rely on standard libraries which provide APIs to manipulate complicated data structures or control concurrent behaviors. These APIs usually come with a specification. Some of them are propositions on the input and output while others are temporal policies on the order of using them. Legunsen et al. [92] formalize specifications of standard Java APIs using the RV tool JavaMOP [81] and check whether the specifications are satisfied in 200 in open-source projects. For instance, the *Iterator_HasNext* property states that each call to *next* on an *Iterator* object must be preceded by a call to *hasNext*. The monitor specification of this policy can be expressed as a state machine, as shown in Figure 3. If *next* is called at the state *ready_st*, the monitor detects this violation by transforming into the state *error*.

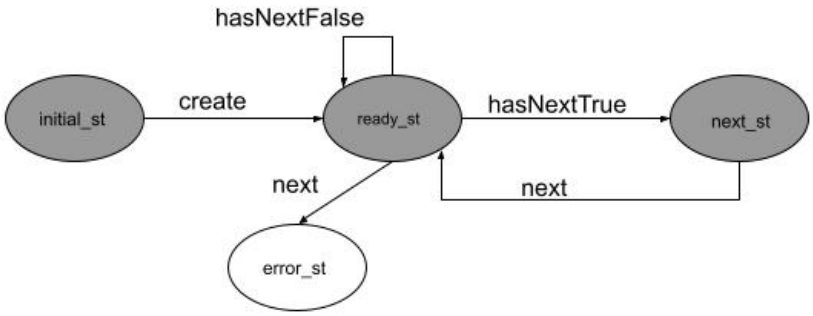

Figure 3: State machine of the *Iterator_HasNext* policy

Case 2: Quality of the tracking application. This example comes from in the RINGS project, led

by BAE Systems. The project focuses on a target tracking application, developed by BAE Systems and continuously evolved over a period of over 15 years. A tracking application receives data from a number of sensors that supply information about observed objects, and contains algorithms that parse sensor inputs and compose observations into tracks, i.e., sequences of points representing position of an object over time. There are a number of metrics that characterize track output quality. These metrics, collected using a sliding window time interval, include average duration of a track observed in a time interval and the number of observations of objects that are not associated with any track. A monitor can be modeled to collect observations, compute metrics and raise an alarm when significant changes are observed.

2.2. RV techniques

2.2.1. RV formalisms

Many RV techniques have been proposed based on temporal logic. Havelund and Roşu [73] present rewriting-based algorithms for past and future time LTL (linear temporal logic) formula. Bauer et al. [22] propose a 3-value logic LTL_3 for runtime verification LTL and TLTL (timed linear temporal logic) formula. A third verdict ? is introduced to denote that the result of an extension of the current prefix of the event trace is not decidable yet. This is particularly useful when monitoring properties that are not pure safety. Some tools such as JPAX [74], DIANA [125] and RiTHM [105] use LTL or extension of LTL as the formalism to describe properties to monitor.

Another powerful formalism is the rule system. The specification is composed of a set of rules. Upon receiving events, rules are evaluated to trigger removal of old rules and add of new ones. EAGLE [11] is a rule-based monitoring system which are expressive to describe multiple formalisms such as past and future time LTL, interval logics and extended regular expressions. To improve efficiency and usability, RuleR [13] is presented to describe properties as executable low-level rules. Several tools and techniques have also been derived such as LogScope [12], TraceContract [10], LogFire [71] and data automata [70].

Automata-based techniques directly describe how properties are monitored and suitable for describ-

ing low-level behaviors. In MaC (monitoring and checking) [87], MEDL (meta event description language) is used to describe monitoring requirements in the form of state transitions triggered by events. Larva [40] uses DATE (dynamic automata with timers and events) [39] to describe properties. Aktug and Naliuka [3] present ConSpec, an automata-based policy language whose semantics is described as security automata [122]. Its specifications can be generated into inline monitors for recognizing security properties [4]. MOP [101] supports using FSM (finite state machine) to describe properties. Reger et al. [115] present MARQ to monitor Java programs, which uses QEA [14] as the formalism to specify parametric properties.

2.2.2. Monitor deployment

Monitors may be deployed synchronously or asynchronously with the target system. Synchronous monitors block execution of the target system until validity of an observation is confirmed. Synchronous instrumentation can be implemented using AOP. While suitable for safety- and security-related contexts, synchronous monitoring might sometimes incur high execution overhead for the target system when execution of the monitoring logic is time-consuming. Moreover, when monitoring multiple programs running in different processes, synchronization is not realistic. In this case, asynchronous deployment comes into play, which is usually implemented by communication middleware [132] or shared buffer [85]. The downside of asynchronous monitoring may come from several aspects: 1) it is difficult to locate the point where the violation happens; 2) the overhead of asynchronous monitoring depends on the underlying communication mechanism, which is harder to predict and may require more efforts to optimize; and 3) timing properties are harder to check when the clock relation between the target system and the monitor cannot be described accurately. Consequently, flexible deployment of monitors is desirable for a RV technique to support different monitoring requirements in different environments. Thanks to the separation between the monitor specification and deployment, many RV techniques such as MOP and Larva support generating both synchronous or asynchronous monitors. Some existing techniques also support hybrid monitoring. Colombo et al. [41] propose an architecture allowing for switching between synchronous and asynchronous monitoring. PolyLarva [42] supports hybrid monitoring of Java programs. Val-

our [7] supports dynamic instantiation and hybrid monitoring. detectEr [59] is an actor-based [2] RV tool for monitoring distributed Erlang programs. A hybrid instrumentation technique used by detectEr is presented by Cassar and Francalanza [32] to dynamically switch between synchronous and asynchronous monitoring, which would reduce the overhead by minimizing the synchronous instrumentation while ensuring timely detections.

2.2.3. Decentralized and distributed runtime verification

Design and implementation of component based systems is hard, especially when modules are deployed in asynchronous environments. Many studies have been proposed to describe properties and generating monitors for these systems. According to whether there exists a global clock between monitors and the target program, there are two categories: decentralized and distributed monitoring [121].

In decentralized monitoring, a monitoring requirement is fulfilled by coordination between a set of monitors. All monitors share the same clock so that a global order between events can be determined. Bauer and Falcone [21] propose a decentralized LTL monitoring algorithm. There are multiple processes which generate different predicates for a formula. To monitor the whole formula locally, the algorithm generates a monitor for each process. Monitors communicate with each other using synchronous communication. At each global tick, each local monitor decides whether a verdict can be generated. If not, it will request results of predicates from other processes. Falcone et al. [54] further generalize the algorithm to support monitoring regular language in a decentralized way. Automata is used to specify local monitors. At each step, the local monitor either gets the update from the component to which it attaches or receives state updates from other monitors. If the information is not enough, it will send state update to other monitors. Colombo and Falcone [38] present another decentralized algorithm for LTL monitoring, which forms a hierarchical network of local monitors according to the structure of the formula statically. During execution, the state of a formula is updated after receiving the update on its sub formulas. El-Hokayem and Falcone [51] present algorithms for monitoring decentralized specifications where monitors can be attached to various components.

Different from decentralized monitoring, distributed monitoring does not assume existence of a global clock. Mansouri-Samani and Sloman [100] present a rule-based language to monitor distributed systems. Given a maximum communication delay, the framework can delay the application of rules and reorder events. Sen et al [125] present a distributed algorithm for monitoring PT-DTL (past time distributed temporal logic), which extends of past time LTL by adding operators to specify formula or values to be computed remotely by other nodes. An algorithm for synthesizing distributed monitors is presented, which relies on knowledge vectors. Each process contains a copy of the vector. The monitor can always obtain the newest evaluation from other processes by knowledge vectors. Francalanza et al. [60] present mDPI, a location-aware π-calculus [102] extension for monitoring distributed systems. Multiple forms of monitoring strategies are presented and evaluated in which one method use migration of monitors to support dynamic architectures. Zhou et al. [141] present DMaC (distributed MaC), which extends MaC system with declarative networking to support checking safety properties of network protocols. The protocol is expressed as NDlog (network datalog) rules [98].

2.2.4. Correctness and verification of runtime monitors

Trustiness of runtime monitors is a vital issue because they are responsible for generating warning and recovery actions correctly when the properties are violated. There are multiple aspects of correctness. If a specification is modeled from a property, guaranteeing that the specification correctly describes the property is important. Some sophisticated monitoring algorithms have been proposed to improve efficiency but correctness of the algorithm must be proved. For some techniques, specifications are transformed into executable monitor code to be inlined with the target program. Verification of compliance between the implementation and the specification is necessary. Representative work on verification on runtime monitors are introduced below.

Laurent et al. [89] present a model checking framework to verify correctness of specifications written in Copilot language [109]. Correctness of a monitor specification is described using invariant properties. A k-induction based model-checker is then applied to check whether the invariant is preserved. The method is useful when the specification is complicated and some invariants should

be discovered and verified to improve trustiness of the specification. However, it requires efforts to write the correct invariants.

Blech et al. [29] present a framework for certified RV using Coq theorem prover [25]. The paper states correctness of monitors from three aspects: instrumentation, integration and compliance of the property specification. The system, system with instrumentation and system with monitor are described as functions to the set of traces. The instrumentation correctness is defined on the equivalence between on the original and instrumented system with respect to projections of system states and concrete events. The integration correctness is defined with respect to non-interference between the monitor and the instrumented system. Monitor correctness is defined on the equivalence of the language. The paper further explores the issue of compliance for the properties described as regular expressions. Both regular expressions and monitor functions are formalized in Coq and preservation of a simulation relation between them is proved inductively. The corresponding relation between concrete and abstract events are also proved.

For formalization and correctness proof of monitoring algorithms, Schneider et al. [123] formalize the monitoring algorithm of MFOTL (metric first order temporal logic) using Isabelle/HOL proof assistant [107] and prove its correctness by establishing an invariant to preserved at each step of the algorithm and verifying that the verdicts generated reflect MFOTL's semantics.

A lot of RV techniques support automatic synthesis of monitors from the property specification and the correctness of synthesis algorithm should be verified. For instance, correctness of the synthesis algorithm for generating automata-based monitors have been proved [62, 124]. Francalanza and Seychell [59] present an automated procedure for synthesizing concurrent monitors for Erlang programs from the property expressed using a subset of HML (Hennessy Milner logic). Due to non-determinism of execution of concurrent programs, correctness of a monitor is defined based on the property violation of a system with respect to an execution. Then, correctness of the synthesis is proved from aspects of violation detectability, detection preservation and monitor separability.

Mitsch and Platzer [103] present ModelPlex, a RV framework for CPS systems. The method as-

sumes a verified model of a CPS system of which the non-determined number of executions preserves a post-condition given a pre-condition. Soundness and computability of the synthesis algorithm are proved.

Finkbeiner et al. [56] propose a verifying compiler that can generate verification conditions as the correctness criteria along with the Rust implementation of Lola monitors [44], which are then proved by the Viper toolkit [104]. Functional correctness, memory safety and termination can be proved in this framework.

2.2.5. Parametric monitoring

Software systems often expand in the sense of data or control by creating new instances of data structure, thread or process. Observations (events) generated from these dynamically scalable structures are attached with identities as attributes of events to distinguish them. Parametric properties are defined on traces of events with attribute bindings. Due to their dynamic nature, parametric properties are hard to be verified statically [34]. Many RV techniques support monitoring parametric properties [75]. Several formalisms of temporal logic have been proposed for parametric monitoring such as JLO (Java logical observer) [130], LTL-FO$^+$ [67], LTL^{FO} [23], MFOTL [18] and monitor modulo theories [46].

Goubault-Larrecq and Olivain present Orchids [65], an intrusion detection tool. Monitors can by dynamically spawned reacting to possible beginnings of attacks. Yamagata et al. [134] present a formalism CSP$_E$ for monitoring concurrent systems. Parametric properties are expressed by recursive parametric processes. Extended from Lola, Lola 2.0 [55] supports parameterized templates and dynamic generation of event streams for parametric monitoring of complex security properties in network traffic.

In our work, we focus on parametric monitoring by trace slicing, which partitions a whole trace into sub traces according the attribute values they carry. Non-parametric properties are then checked against the sub traces. Allan et al. [5] first implements trace slicing in Tracematches to support event matching with values of parameters. MOP implements an efficient trace slicing al-

gorithm [34] for parametric monitoring, which has also been used in other tools such as RV-monitor [99], MOVEC [35] and QEA. QEA checks parametric properties using the same trace slicing mechanism with MOP. However, parameters can be quantified by a nested list of universal and existential quantifiers. As a result, QEA is more expressive than MOP in parametric monitoring but less in time efficiency due to the difference in the binding semantics. We will further study this difference in Chapter 4.

There is also work on exploring the relation between specification techniques for parametric monitoring. Reger and Rydeheard [114] present a transformation from QEA to rule-based system and differences between these two techniques with respect to parametric monitoring are highlighted. Reger et al. [113] present a subset of syntactic fragments in first-order temporal logic that are sliceable and transform them into automata for slicing. The core of parametric monitoring is using indexing to access states for an incoming event [72].

2.2.6. Stream processing

Stream processing [8] is a technique to handle data flow at large scale. A stream processing system is usually composed of a set of nodes. On each node, operations such as union, join, filter and arithmetic computation can be specified. Nodes are connected to form a DAG to transform event streams. Compared to traditional batch processing, stream processing can efficiently deal with real-time data and it is suitable to be used in event-driven applications. Many stream processing frameworks, such as Storm [1], Flink [31] and Spark [135] have been proposed. With support of stream operators, properties over data streams can be conveniently described in stream processing programs and the mature framework makes it suitable to handle large-scale data. However, because general stream processing frameworks handle large-scale data in a distributed way, which requires more sophisticated control and brings more runtime overhead, it is not suitable when the target system is not at large scale. To monitor properties over data streams, many stream runtime verification (SRV) techniques have been proposed such as Lola [44] and its successor Lola 2.0 [55], Striver [64], TeSSLa [94], Copilot [109] and Quantitative regular expressions (QREs) [6]. They have different application targets and thus make different assumptions over streams and provide different language

features. For instance, Lola is a DSL for specifying properties for synchronous systems. Each data point comes with a timestamp and different streams are synchronized by a global clock. Lola 2.0 gets rid of the global clock and provides sliding window expressions to aggregate events for time intervals. Moreover, as already mentioned above, dynamic generation of event streams to support large-scale inputs. More complicated properties such as network traffic can be described. QRE is a specification language for complex numerical queries over data streams. Stream composition and arithmetic operators such as sum and average are supported.

2.2.7. Summary

In this section, we have introduced some representative studies on runtime verification. To monitor a variety of properties for different types of systems, existing studies focus on multiple aspects such as design of specification languages, monitoring algorithms, decomposition of specifications and deciding deployment of monitors. However, fulfilling two requirements introduced in Section 2.1 is still challenging for many existing RV techniques. In Chapter 3, we present a new formalism, SMEDL for RV which utilizes automata to specify properties at multiple abstract levels. A complicated monitoring requirement can be decomposed into a set of monitors that can communicate with each other during runtime. Moreover, users can specify deployment of monitors in a flexible way.

To bridge the gap between the monitor specification and the implementation, we propose a method to generate correct-by-construct implementation of monitors using Coq theorem prover and the Fiat framework. In Chapter 4, we present a novel method to use dynamically scalable SMEDL monitor network for parametric monitoring. In Chapter 6, we will also study monitoring timing properties in distributed environments where network delay and clock difference need to be taken into consideration.

2.3. Self-adaptive software

Software adaptation is a technique to enable software to dynamically adjust changes during execution. According to the objective of adaptation, there are four types of goals [119]: self-configuration, self-optimization, self-healing and self-protection. We will first introduce work on general tech-

niques for adaptation, which are divided into two categories, method-based and model-free methods. Then, we present work on runtime enforcement, which uses monitoring techniques to detect violation of properties during runtime and then generates reflexive actions to prevent software failure or malicious attacks.

2.3.1. Model-based methods

The adaptation process contains four steps, monitor, analyze, plan, and execute, which require knowledge of the target system (referred as MAPE-K [82]). In model-based software adaptation, a model provides hints on when and how to apply adaptation actions and guarantees correctness of the adaptation. Both architecture and behavior models can be used in the adaptation.

The rainbow framework [61] uses style-specific software architectures and a reusable infrastructure to support self-adaptation. An architecture model contains properties to be maintained during execution. After a violation is observed and analyzed, repairs are conducted which would change the architecture. MADAM [57] uses architecture models to realize software adaptation. The adaptation is implemented as a middleware, which monitors user and system contexts and uses an architecture model to decide whether adaptation is needed and how to apply it. Utility functions are used to decide when to switch between implementations to achieve the adaptation goal. The application variant with the highest utility will be chosen. ActivFORMS [79] is an adaptation technique which implements the MAPE-K loop formally and supports changing goals at runtime. Models of timed automata can be executed by a virtual machine so that no coding is required.

Studies introduced above can support sophisticated adaptation actions. However, a large fraction of software failures, especially ones depending on the environment and concurrent behaviors can be solved by simply rebooting the system. Microreboot [30] uses local recovery to increase the availability of Java-based Internet systems. Compared to global recovery, local recovery is more efficient and potentially has less influence to the user. However, the only type of actions microreboot supports is restarting a subset of components. Sözer et al. [128] propose the Flora framework to support local recovery based on decomposition of component-based system. Components are

partitioned based on the independent relation and the cost to execute restarting operation. During execution, Flora will detect whether failure happens in each partition and only reboots the failing ones.

Easwaran et al. [49] explore a control-theoretic view of software adaptation and present an architecture for steering of discrete event system [111]. The steering architecture contains a system, a monitor and a steerer. The system is modeled as a transition system receiving activation or deactivation signals sent from the steerer. Activation signals carry the actions to be executed in the system and deactivation signals halt the specified actions. Monitors are used to return possible violations that the system may encounter so that the steerer can generate actions to prevent the violation. The communication delay between the system and the controller is handled as a control problem under partial observation.

2.3.2. Model-free methods

Model-free methods do not require systematic knowledge on the system or the environment. ASSURE [126] is a system introducing rescue points that recover from unanticipated problems from known error conditions. Prior to deployment, ASSURE uses fuzzing to discover candidate rescue points. During execution, when a fault is detected, ASSURE analyzes the fault, selects a proper rescue point and then produces a patch. The patch instantiates a rescue point inside the application to avoid the fault. Ares [66] also tries to leverage existing error handlers to recover from unexpected errors. It intercepts the exception handling of the underlying JVM. A synthesizer generates a set of candidate solutions by analyzing the exception type and the call stack. Java PathFinder then analyzes candidates and ranks them to choose the most viable one to patch. Nielebock [106] presents an API-specific program repair mechanism. The method utilizes API information from the erroneous code to search for API usage patterns from which patches can then be generated. Kim et al. [86] extend MaC with a feedback capability. Steering specification includes the target objects and actions to be executed. When the system deviation is detected by the runtime checker, actions are invoked and transferred to the target system and executed when the system is ready.

2.3.3. Runtime enforcement

Runtime enforcement (RE) extends runtime verification with the ability to transform from an input event trace to an output trace that follows the specification. Some EMs (enforcement monitors) are utilized as the interface to generate property-compliant data for two systems communicating with each other using events. Reference monitors [68], on the other hand, intercept actions generated from the target system and generates actions to change the state of the program directly to prevent violation of the property. There have been many studies focusing on multiple perspectives of RE from specifications to principles of RE. Some representative ones are introduced below.

The initial formalism of RE is security automata (SA) [122], which is a büchi automata with partial transition functions. When a security automata receives an event, it checks whether there is a defined transition. If so, the transition is executed and the target system can continue execution. Otherwise, the target system will be halted. SA needs to memorize potentially unbounded execution history. Fong [58] presents SHA (shallow history automaton) which only tracks what events have been received without recording the order among them. Although strictly less expressiveness than SA, SHA requires less memory and can still enforce some famous security policies such that the Chinese Wall policy. Sometimes, security related method calls to be enforced will return value, Ligatti and Reddy [95] present MRA (mandatory result automata) which is used as an interface between the untrusted application and the executing system. The MRA sends valid calls from the application to the system and returns valid results generated from the system back to the application. Under certain environments where MRAs can explicitly observe termination of the application and the executing system always return results, MRAs can enforce action-life properties.

SA can only enforce safety properties. Ligatti et al. [96] present edit automaton which supports suppressing and inserting actions. Renewal properties can be enforced by edit automata [97]. Renewal properties contain all safety and certain liveness properties. Falcone et al. [52] present GEM (generalized enforcement monitors) with finite control states and a memory device. Store and dump operations are supported to memorize events and re-insert them when necessary. GEM can enforce response properties which expect good things should happen infinitely often. Response properties

coincide with renewal properties [53]. Bielova and Massacci [28] study enforcement of iterative properties. Informally, a property is iterative if concatenation of two property-compliant traces still satisfy the property. They present iterative suppression automata to remove invalid subparts of the trace.

RE mechanism has been used to implement security policies. Bauer et al. present Polymer [24], a system using edit automata to specify and compose security policies for Java programs. Another application of runtime enforcement lies in the usage control policy. El-Harake et al. [50] apply enforcement techniques to block advertisements embedded in the Android applications. Riganelli et al. [117] introduce proactive library for API usage in the Android ecosystem. At the development phase, developers model usage policies as edit automata, which will be compiled into policy enforcer which guarantees correct API usage by intercepting API calls from applications to the libraries.

Since enforcement mechanisms may directly change behavior of the target system, correctness is an important issue. Current studies focus on two aspects. The first aspect is defined on the specification, which says *soundness* and *transparency* must be guaranteed by an EM [96]. Soundness means the output of an EM must comply with the property while transparency means a correct input trace should not be altered. For EMs of safety properties, two principles can be satisfied directly since the prefix of a trace must also satisfy the property. For properties having invalid prefixes, we want to output a correct trace that is as close as to the input trace. The notion of precise and effective enforcement are defined. Precise enforcement requires that if the input trace is valid, the output trace must not be modified. Effective enforcement is based on equivalence relation between executions. One instantiation of equivalence relation is the longest valid prefix [52]. Bielova and Massacci [27] distinguish different ways of executing suppression and insertion actions. An *All-Or-Nothing* automaton stores invalid prefix until the stored trace complies with the property while a late *automaton* only outputs some valid prefix. They also discuss the case where soundness and transparency are too strict to define bad traces [26]. Distance between two traces are used to measure closeness between two traces and boundedness and predictability are defined as a weaker notion of

soundness and transparency.

The second aspect is the gap between the specification and the implementation. Several studies have been proposed to verify correctness of in-line monitors. Hamlen et al. [69] present MOBILE, a framework to certify monitors on Microsoft .NET architectures using type theory. The method assumes a security property is defined correctly but the rewriter, which is responsible for generating the implementation, cannot be trusted. The compliance of property is checked by type-checking rules defined upon bytecode programs. Aktug et al. [4] present a proof-carrying monitor inlining mechanism for Java Virtual Machine. The inlining correctness is described using the annotations in the form of Hoare logic. A ghost monitor represents the policy specification, which is transformed into checks before and after security-related calls. The proof of state synchronization between the ghost monitor and the in-line monitor is built locally as pre- and post-condition for each relevant program points. Using the verification condition checker, the adherence proof is validated. Sridhar and Hamlen [129]. design and implement a model checking framework for the correctness of in-line reference monitor. The method assumes that the in-line monitor is generated and integrated into the ActionScript bytecode programs by an untrusted third-party. The verifier non-deterministically explores the untrusted code and obtains an abstract machine describing the behavior of the monitor. The model checker then verifies whether the abstract states comply with the security property. The soundness and convergence of the abstract machine is proved.

2.3.4. Summary

Runtime enforcement has been used to either prevent the system from property violation or generate reflexive fixes. Compared to general techniques which require models or existing error handling mechanism, RE is light-weighted and easy to validate. However, even though existing techniques have provided ways to check transparency and soundness of a formalism and compliance between the specification and the implementation, there is little work on interference between the target system and the implementation of adaptation actions. In Chapter 7, we will extend SMEDL to support reflexive adaptation actions and present an initial work on verifying correctness of adaptation actions with respect to the target system.

CHAPTER 3 : Scenario-based Meta Event Description Language

This chapter presents Scenario-based Meta Event Language (SMEDL) as a formalism to express monitoring logic. The formalism is divided into two parts: single monitor specifications and architecture descriptions. A single monitor specification is composed of a set of EFSMs which share data states and synchronize with each other using internal events. In Section 3.1, we formally define the specification language and propose a method to generate correct-by-construction implementation of single monitors. By using single monitors, many properties can be expressed from ones describing high-level temporal relations to ones involving primitive computations and state updates. To further conveniently describe and efficiently check properties of component-based systems that may scale during runtime, in Section 3.2, we use single monitor specifications as building blocks to construct a more complicated monitoring system as a monitor network. An architecture description specifies flows of information between monitors. Monitor deployment and communication between monitors can also be described. A single monitor specification is extended to carry identities to support dynamic instantiation of instances on demand. The semantics of dynamic instantiation of monitor instances and synchronized collaboration of monitors will be proposed.

To make this chapter more self-contained, we first introduce terminologies, notations and definitions. Sets are denoted by single uppercase letter or strings starting with a capital letter. The power set of S is denoted as 2^S or $\mathscr{P}(S)$. Lowercase letters or strings are used to denote variables or values. Symbols may be attached with subscripts of natural numbers. The symbol $\rightarrow$ will be overloaded when defining a function space, mapping or transition. The symbol $\rightharpoonup$ denotes a partial function. When denoting the internal structure of a set, we will use curly brackets. Comprehension notation is used to describe the condition to be satisfied in the set. The tuple is represented using angled brackets or parentheses. We use θ to denote a map, which can be expanded using the notation $[x_1 \mapsto v_1, x_2 \mapsto v_2 ...]$. We use the operation $\theta[x_1]$ to return v_1. The symbol $=$ is used for assignment or equality. The symbol $\equiv$ is used for equivalence relation or definition. Dot notation and subscripts are used to represent the relation between the element and the tuple to which it belongs. For instance, for a tuple $T \equiv \langle A, B \rangle$, the field B of T can be denoted either as $T.B$ or B_T.

Dot notation is also used to concatenate lists and the empty list will be denoted by ϵ.

3.1. Single monitor: design, semantics and correct-by construction code generation

A SMEDL single monitor specification is an entity that can be used to describe a complete monitoring requirement or as a building block to form a monitor network to describe more complicated properties. By using a higher-level specification language than common programming languages, monitors can succinctly describe what monitors should do: implementation details are crucial for performance, but they can be determined separately, along with a proof of preservation of semantics between the specification and implementation. One challenge is to design semantic rules that can be used smoothly in the specification, while remaining amenable to refinement so that such preservation can be proved without excess difficulty.

Additionally, to generate a correct monitor, we have to ensure that the monitor specification is well-formed. This thesis focuses on two properties: *responsiveness* and *determinism*. Informally, responsiveness means the monitor always returns a result when fed with an event, which is important because if a monitor gets stuck during the execution, it cannot receive events from the system and catch property violations. Moreover, a synchronous monitor that gets stuck or aborts will directly influence execution of the target system. Determinism ensures that the monitor always produces the same output given the same input and the state. The code generation process needs to detect and reject any "bad" monitor specifications that may behave non-deterministically during runtime.

In this section, we first present the definition of single monitors. Then, we present an operational semantics for SMEDL monitors written in a relational way which ensures that the functionality of a specification and its implementation are separated. Based on the semantics, we define a predicate on the definition of SMEDL monitors to ensure responsiveness and determinism. Finally, we present a method for generating correct-by-construction implementations of single monitors. Our method is based on Fiat [47], a deductive synthesis framework embedded in the Coq theorem prover. Users start by embedding their DSL (domain specific language) into Coq, so that each DSL program is understood as a mathematical description of the set of results it may return. Using stepwise refine-

ment, each program can be translated into a correct-by-construction executable implementation. A case study illustrating the usability of the code generation will be presented[1].

3.1.1. Definitions

A SMEDL monitor is a reactive entity interacting with the environment by receiving and generating events. A monitor is composed of a set of *scenarios*. Each scenario is an EFSM reacting to events. Scenarios interact with each other using shared state variables or by triggering execution of other scenarios through raised events. Each transition is labeled with a triggering event and attached to a boolean expression as the guard condition and a list of actions to be executed after the transition. Primitive data types such as integer, float and boolean types and arithmetic and logical operations are supported in SMEDL. The definition of the monitor is given below.

Definition 1 (Monitor) *A monitor M is a triple $\langle V, E, S\rangle$, where V is a set of state variables; E is a set of event declarations and S is a non-empty set of scenarios.*

Definition 2 (Event declaration) *An event declaration is a triple $\langle evName, attributeTypes, event\ Type\rangle$ where $evName$ is the name of the event, $attributeTypes$ is a list of data types for the attributes of the event; $eventType$ is an enumeration of three values: $imported$, $exported$ and $internal$ to denote how the event is used in the monitor specification.*

Imported events are received from the environment to trigger execution of the monitor; exported events are raised within the monitor and sent to the environment; internal events are also raised within the monitor but are only seen and processed within a given monitor.

Definition 3 (Scenario) *A scenario of a monitor M is an EFSM $\langle St, \Sigma, \iota, \psi, F\rangle$, where St, F, ι are respectively a set of states, a set of final states and the initial state; Σ is the alphabet—events that can trigger the transitions of M, which is a subset of $M.E$; ψ is a set of transitions. All scenarios are assumed to be complete with respect to transitions.*

Definition 4 (Transition) *A transition is a 4-tuple $\langle qsrc, qdst, ev, \mathcal{A}\rangle$ where $qsrc$ and $qdst$ are the*

[1] https://github.com/PRECISE/smedl-fiat-code

source and target state of the transition; ev is an event instance that triggers the transition; $\mathcal{A}$ is a list of actions to be executed after the transition.

Definition 5 (Event Instance) *An event instance of a transition tr is a 3-tuple $\langle event, eventArgs, g \rangle$ where $event$ is a reference to the event declaration, $eventArgs$ is a list of variables that are local to tr and g is the guard condition of tr.*

For each event instance *ev* of the transition *tr* defined in the scenario *sce*, *ev.event* $\in$ *sce.*Σ. Moreover, the data type of local variables in *eventArgs* are implicitly declared in *ev.event.attributeTypes*. We will use the notation $qsrc \xrightarrow{ev\{\mathcal{A}\}}_{ev.g} qdst$ to denote the transition $\langle qsrc, qdst, ev, \mathcal{A} \rangle$.

Supported actions are state update and raising events. In state update actions, state variables or local variables can be updated by normal arithmetic or logical operations. Events to be raised must be an internal or exported event.

Definition 6 (State Update) *A state update action is a tuple $\langle v, expr \rangle$ where v is the variable to be updated and $expr$ is the expression computing the value to be assigned to v.*

Definition 7 (Event Raising) *A event raising action is a tuple $\langle e, exprList \rangle$ where e is the name of the event to be raised and $exprList$ is a list of expressions to be computed and bound to the attributes of e.*

To give an illustrative example of a single monitor, the SMEDL specification for *Iterator_HasNext* of *Case 1* in Section 2.1 is given below. Note that self-looping transitions are omitted. Three events *create*, *next* and *hasNext* represent operations of creating an iterator, calling to the *next* and *hasNext* API. The attribute *b* is the return value of *hasNext*. If the call to *next* is made at the *ready_st* state, the monitor will raise an exported event *error* to indicate the violation of the API policy. The concrete syntax of single monitors is given in Appendix A.1.1.

```
object IteratorHasNext
    events:
        imported create();
        imported next();
        imported hasNext(int);
        exported error(int);
    scenarios:
       main:
           init_st -> create() -> ready_st;
           ready_st -> hasNext(b) when (b == 1) -> next_st;
           next_st -> next() -> ready_st;
           ready_st -> next() {raise error();} -> error_st;
```

3.1.2. An Operational Semantics of SMEDL

This section presents an operational semantics for a single monitor, which is represented as transitions between *configurations*.

Definition 8 (Configuration) *A configuration of a monitor M is a five-tuple $\langle MS, DS, PD, EX, SC\rangle$ where*

- *MS is a mapping from $M.S$ to their current states;*
- *DS is a well-typed function $M.V \rightarrow$ Val where Val is the set of values;*
- *PD is a set of pending events to trigger transitions within M in the current macro-step;*
- *EX is a set of raised exported events;*
- *SC is a set of scenarios that have been executed during the current macro-step.*

Each configuration *conf* relates to a monitor *M*, denoted as $conf_M$. The subscript is omitted whenever the context is clear. Each element in *PD* and *EX* is a *raised event*, which is defined below. For both raised events and event instances, we abuse the word *type* to state their relation with the corresponding event declaration. For a well-typed raised event e, all values of *e.valList* should be compatible with types specified in *e.event.attributeTypes*.

Definition 9 (Raised Event) *A raised event is a tuple* $\langle event, valList \rangle$ *where event is a reference to the event declaration and* $valList$ *is a list of values.*

To trigger a transition, its event instance ei must be matched with a raised event e in PD and local variables $ei.eventArgs$ are bound to $e.valList$. A function $bind(ei, e)$ is defined to generate a mapping θ from $ei.eventArgs$ to corresponding values in $e.valList$. For succinct representation, we will use the terminology *event* to denote the concept of *event declaration*, *event instance* and *raised event*.

Definition 10 (bind function) *bind(ei, e) returns a mapping* θ *where ei.event = e.event* $\wedge\ \theta[n_j] = v_j$ *for all variables* $n_j \in ei.eventArgs$ *and its corresponding value* $v_j \in e.valList$ *at the same position in the list.*

When an imported event is sent to a monitor, state transitions within the monitor are triggered. Actions attached to transitions may raise internal or exported events. Internal events are used to trigger further transitions in other scenarios. After all triggered transitions are completed, exported events are output and the monitor waits for the next imported event. This process is denoted as three levels of transitions. At the highest level, a *macro-step*, denoted as $conf \overset{e}{\rightharpoonup} conf'$, represents that the evolution of a monitor from $conf$ to $conf'$ by an imported event e. A macro-step is constructed by chaining a series of consecutive *micro-steps*, denoted as $conf \overset{e}{\hookrightarrow} conf'$. Each micro-step is a synchronous composition of a set of *scenario-steps* on scenarios with the same triggering event. At the lowest level, each scenario-step, denoted as $conf \overset{e}{\longrightarrow} conf'$, is applied to a scenario. Each scenario can take at most one scenario-step in each macro-step so that there are no infinite interactions between scenarios. For constructing three types of transitions, *basic rule*, *synchronoy rule* and *chain merge rule* are proposed below. The semantic rules have been encoded in Coq. For succinct representation, type checking rules, evaluation of expressions and sequence rules performed on a list of actions are omitted.

Basic rule. The basic rule is applied to a scenario whenever a transition is triggered by a pending event. In the definition below, the scenario performing the transition is denoted as *mh*, *conf* denotes

the configuration before applying the rule, and $conf'_{mh}$ denotes the configuration after applying the rule on *mh*. The union operator $\cup$ is overloaded for generating the union of two mappings. The function $eval(expr, m)$ evaluates $expr$ given a mapping m from variables to values.

$$\frac{\begin{array}{c} tr : s1 \xrightarrow{ei\{a\}}_c s2 \quad tr \in mh.\delta \\ MS_{conf}(mh) = s1 \quad mh \notin SC_{conf} \\ e \in PD_{conf} \;\; \theta = bind(ei, e) \\ DS_{ex} = DS_{conf} \cup \theta \quad eval(c, DS_{ex}) = true \\ \langle DS', PD', EX' \rangle = updateConfig(a, DS_{ex}) \\ conf'_{mh} = \langle MS_{conf}[mh \mapsto s2], DS', (PD_{conf} \setminus \{e\}) \cup PD', EX_{conf} \cup EX', SC_{conf} \cup \{mh\} \rangle \end{array}}{conf \xrightarrow{e} conf'_{mh}}$$

When receiving an event e, *tr* is the enabled transition in *mh* from $s1$ to $s2$ by ei which matches e. To execute the transition, the following preconditions must be satisfied: 1) current state of *mh* is *s1* and 2) c, the guard condition of tr, is evaluated to true given the current data state which is the union of *DS* and the mapping θ from local variables in ei to attribute values stored in e, 3) *e* is in *PD*, and 4) *mh* is not in *SC*.

When *tr* is taken, *mh* transitions to $s2$ and is put into *SC* and *e* is removed from *PD*. By executing actions in a under the context DS_{ex} (represented by the function *updateConfig*), *DS* is updated to *DS'*; and raised events are respectively added to *PD* and *EX* according their types.

Synchrony rule. One or more scenarios may be enabled by a triggering event from the same source configuration. The basic rule creates new configurations for each scenario by taking these transitions. The synchrony rule then combines scenario's resulting configuration into a new configuration. Combination of two configurations $conf_1$ and $conf_2$ under the origin configuration *conf* is defined below.

- $\forall mh \in S$,

$$MS_{conf'}(mh) = \begin{cases} MS_{conf_1}(mh)\ , MS_{conf_1}(mh) = MS_{conf_2}(mh) \\ MS_{conf_1}(mh)\ , MS_{conf_1}(mh) \neq MS_{conf}(mh) \\ MS_{conf_2}(mh)\ , MS_{conf_2}(mh) \neq MS_{conf}(mh) \end{cases}$$

- $\forall v \in V$,

$$DS_{conf'}(v) = \begin{cases} DS_{conf_1}(v),\ DS_{conf_1}(v) = DS_{conf_2}(v) \\ DS_{conf_1}(v),\ DS_{conf_1}(v) \neq DS_{conf}(v) \\ DS_{conf_2}(v),\ DS_{conf_2}(v) \neq DS_{conf}(v) \end{cases}$$

- $PD_{conf'}$= $PD_{conf_1} \cup PD_{conf_2}$

- $EX_{conf'}$= $EX_{conf_1} \cup EX_{conf_2}$

- $SC_{conf'}$=$SC_{conf_1} \cup SC_{conf_2}$

The synchrony rule given below is used to create a *micro-step*. *confs* is the set of target configurations obtained from the basic rule given a source configuration *conf* and an event *e*. *MergeAll* combines each configuration in *confs* into a new configuration by repeatedly combining configurations pairwise. The micro-step from c to c' by e is denoted as $c \overset{e}{\hookrightarrow} c'$.

$$\frac{confs = \{conf_{mh} | conf \xrightarrow{e} conf_{mh}\}}{conf \overset{e}{\hookrightarrow} MergeAll(confs)}$$

Chain merge rule. The objective of the chain merge rule is to construct a macro-step, which is the transitive closure of micro-steps. Case (1) shows that a micro-step triggered by an imported event is the basic case. The corresponding source configuration is denoted as an *initial* configuration, which is defined below. The inductive case is shown in case (2). Note that all internal events are forgotten because they are not observable from the outside of the monitor. There is no restriction on how to choose the next event from pending events of *conf'*. Note that integer subscripts are attached in the chain merge rule to indicate the number of micro-steps from the initial configuration to the current

configuration, which will be used to prove the *responsiveness* property below.

$$\frac{conf \overset{e1}{\hookrightarrow} conf'}{conf \overset{e1}{\rightharpoonup}_1 conf'} \; e1.event.eventType = imported \; (1)$$

$$\frac{\begin{array}{c} conf \overset{e1}{\rightharpoonup}_n conf' \\ conf' \overset{e2}{\hookrightarrow} conf'' \end{array}}{conf \overset{e1}{\rightharpoonup}_{n+1} conf''} \; (2)$$

3.1.3. Towards a well-formed monitor specification

To bridge the gap between the semantic rules and the monitor implementation, we need to overcome some challenges. Firstly, the basic and the synchrony rule are encoded as partial functions in Coq, which means some restrictions are necessary to make sure their application always succeeds. Secondly, the chain merge rule is defined relationally because it does not specify which event to choose from *PD* to trigger the next micro-step, nor does it guarantee termination during the combination of micro-steps. To derive a computable version, which must terminate because of restrictions in Coq, the chaining process must terminate normally each time given an imported event. Moreover, all possible implementations of the chain merge rule must be equivalent in the sense of verdicts they produce given the same input and the same state. To summarize, a monitor must satisfy two properties, *responsiveness* and *determinism*. To describe a state that a monitor can stay after a macro-step, we define the concept of an *initial* configuration and a *final* configuration.

Definition 11 (Initial Configuration) *A configuration conf is an initial configuration if 1) $SC_{conf} = \emptyset$ and 2) $PD_{conf} = \{e\}$ where e is an imported event.*

Definition 12 (Final Configuration) *A configuration conf is a final configuration if 1) $SC_{conf} \neq \emptyset$ and 2) $PD_{conf} = \emptyset$.*

Definition 13 (Responsiveness) *A monitor M is responsive iff for any two of its configurations $conf_M$ and $conf'_M$ and an imported event e that satisfies the relation $conf_M \overset{e}{\rightharpoonup}_n conf'_M$, if M cannot take any micro-step from $conf'_M$, then $conf'_M$ is a final configuration and n is equal to or less than*

$|M.S|$.

Definition 14 (Determinism) *A monitor M is deterministic iff given the configuration* $conf_M$, $conf'_M$ *and* $conf''_M$, *if* $conf_M \xrightarrow{e} conf'_M$, $conf_M \xrightarrow{e} conf''_M$ *and both* $conf'_M$ *and* $conf''_M$ *are final, then* $conf'_M = conf''_M$.

As a side note, we can make informal connection from the evolution of configurations to the theory of *abstract rewriting system* (ARS) [88] where configurations and micro-step are the alphabet and the binary relation of an ARS. Responsiveness and determinism correspond to the *strong normalizing* and *confluent* property. Although we will not formalize this connection in this work, some idea will be used below to prove the determinism property of a monitor.

Table 1 lists predicates on the syntactic structure of a monitor, which are divided into three categories indicating which parts of the semantic rules are influenced. Note that subscript is used to denote the relation between a tuple and its fields.

Table 1: Predicates for well-formedness

Classification	Name	Definition
Scenario-step level	P1	$\forall s \in S_M,\ \forall tr_1\ tr_2 \in \delta_s, qsrc_{tr_1} = qsrc_{tr_2} \wedge ev_{tr_1}.event = ev_{tr_2}.event \Rightarrow\ g_{ev_{tr_1}} = \neg g_{ev_{tr_2}}$
Micro-step level	P2	$\forall v \in V_M\ , \forall s1\ s2 \in S_M\ , updateVar(v, s1) \wedge updateVar(v, s2) \Rightarrow \Sigma_{s1} \cap \Sigma_{s2} = \emptyset$
Macro-step level	P3	$\forall e \in E_M,\ eventType_e = imported \vee eventType_e = internal \Rightarrow \exists s, s \in S_M \wedge e \in \Sigma_s$
	P4	$\forall e\ e1\ e2 \in E_M, e1 \neq e2 \wedge e \Uparrow_M e1 \wedge e \Uparrow_M e2 \Rightarrow \neg\exists s, s \in S_M \wedge e1 \in \Sigma_s \wedge e2 \in \Sigma_s$
	P5	$\forall e\ e1 \in E_M, eventType_e = imported \wedge e \neq e1 \wedge e \Uparrow_M e1 \Rightarrow \neg\exists s, s \in S_M \wedge e \in \Sigma_s \wedge e1 \in \Sigma_s$
	P6	$\forall e \in E_M, \forall\ s1\ s2 \in S_M, raiseEv(s1, e) \wedge raiseEv(s2, e) \wedge s1 \neq s2 \Rightarrow \neg\exists e' \in E_M, trigSce(s1, e') \wedge trigSce(s2, e')$
	P7	$\forall e \in E_M,\ s \in S_M,\ stp \in \delta_s,\ noDuplicatedRaise(e, s, stp)$
	P8	$\forall e1\ e2 \in E_M, \forall v \in V_M, \exists e, noDependency(e, e1, e2) \wedge updateVarEv(v,e1) \Rightarrow \neg updateVarEv(v,e2) \wedge \neg usedVarEv(v,e2)$

P1 guarantees that exactly one transition is triggered for a scenario during the application of the basic rule, by an event from the alphabet for that scenario. *P2* guarantees that when applying the synchrony rule to construct a micro-step, scenarios that share the same triggering event never update the same variable. $updateVar(v, sce)$ means that variable *v* is updated by actions from the transitions of scenario *sce*.

The tricky part is that all pending events must be consumed at the end of each macro-step, i.e. there are no pending events when all available scenarios have finished execution and that the execution of a monitor never gets stuck because of a mismatch between enabled scenarios and pending events. *P3* guarantees that all imported events or internal events can trigger execution of some scenarios. *P4* and *P5* ensure that imported or internal events that may be raised in the same macro-step cannot directly trigger execution of the same scenario. $e \Uparrow_M e1$ means that $e1$ is raised by the actions of transitions transitively triggered by *e*. *P6* and *P7* guarantee that in each macro-step, an internal event cannot be raised multiple times. $raiseEv(sce, e)$ means that the actions of transitions defined in *sce* contain raising *e*. $trigSce(sce, e)$ means that *e* may transitively trigger a transition of *sce*. $noDuplicatedRaise(e, sce, stp)$ means that e can only be raised once in *stp* of *sce*.

The chain merge rule does not specify an order for the chaining of micro-steps. If a monitor is not well defined, the result of a macro-step could be non-deterministic. *P1* ensures scenario-level determinism. *P4* to *P7* also prevent some behaviors that may lead to non-determinism. We define a proposition $noDependency(e, e1, e2) \equiv eventType_e = imported \wedge e \Uparrow_M e1 \wedge e \Uparrow_M e2 \wedge \neg e1 \Uparrow_M^e e2 \wedge \neg e2 \Uparrow_M^e e1$. This means that $e1$ and $e2$ may be raised in the macro-step triggered by imported event *e*, and that during this macro-step, $e1$ can not transitively raise $e2$, and vice versa. *P8* guarantees that updating a state variable is mutually exclusive. $updateVarEv(v, e)$ and $usedVarEv(v, e)$ respectively mean that *v* may be updated and used in any actions transitively triggered by *e*.

We use the notation $Pi(M)$ to represent that a monitor M satisfies predicate Pi. A well-formed monitor satisfies the eight predicates defined above, *Wellformed(M)* $\equiv \bigwedge_{1 \leq i \leq 8} Pi(M)$.

Given a monitor that is well-formed, and starts execution from an initial configuration, we can now prove that it always reaches a final configuration deterministically within a bounded number of micro-steps, as described in Theorem 15 and Theorem 16 below:

Theorem 15 *A well-formed monitor M is responsive.*

Theorem 16 *A well-formed monitor M is deterministic.*

To prove Theorem 15, we need to first prove that the number of micro-steps taken within a macro-step is bounded. Because each scenario can only transition once during each macro-step, and at least one scenario executes in each micro-step, the number of micro-steps to be taken is bounded by the number of scenarios in the monitor. So we first prove that $|SC_{conf}|$ strictly increases in a micro-step.

Lemma 17 *Given two configurations $conf$ $conf'$ and an event e, if $conf \overset{e}{\hookrightarrow} conf'$, then $|SC_{conf}| < |SC_{conf'}|$.*

With Lemma 17 and the fact that SC_{conf_M} is a subset of $|S_M|$, we can prove that the number of micro-steps taken by a well-formed monitor in a macro-step is bounded by the number of scenarios:

Lemma 18 *Given a well-formed monitor M, two of its configurations $conf_M$ and $conf'_M$ and an imported event e, if $conf_M \overset{e}{\rightharpoonup}_n conf'_M$, then $n \leq |S_M|$.*

Next we need to prove that a well-formed monitor cannot be stuck in a non-final state:

Lemma 19 *Given a well-formed monitor M, two of its configurations $conf_M$ and $conf'_M$ and an imported event e, if $conf_M \overset{e}{\rightharpoonup}_n conf'_M$ and $conf'_M$ is not a final configuration, then M can take a micro-step on all of its pending events from $conf'_M$.*

With the three core lemmas presented above, and other auxiliary lemmas, Theorem 15 can be proved by using the idea of *confluence* in rewriting systems. First, we prove the *diamond* lemma defined below:

Lemma 20 (Diamond) *Given a well-formed monitor M, if $conf_M$ is an initial configuration or*

there exists a configuration $oconf$ *such that* $oconf \xrightarrow{e} conf_M$*, and* $conf_M \overset{e1}{\hookrightarrow} conf1_M$ *and* $conf_M \overset{e2}{\hookrightarrow} conf2_M$*, then there exists a configuration* $conf'_M$ *such that* $conf1_M \overset{e2}{\hookrightarrow} conf'_M$ *and* $conf2_M \overset{e1}{\hookrightarrow} conf'_M$*.*

The precondition of Lemma 20 on the source configuration is used to guarantee that the transition is performed from a legal configuration. Then, by induction on the number of micro-steps to be taken by two transition chains, we can prove the *confluence* lemma:

Lemma 21 (Confluence) *Given a well-formed monitor* M*, if* $conf_M \xrightarrow{e} conf1_M$*,* $conf_M \xrightarrow{e} conf2_M$ *, there exists a configuration* $conf'_M$ *such that* $conf1_M \hookrightarrow_* conf'_M$ *and* $conf2_M \hookrightarrow_* conf'_M$*.*

Transition $\hookrightarrow_*$ represents multiple micro-steps. Lemma 21 ensures that if an initial configuration *conf* can transition into two non-final configurations $conf_1$ and $conf_2$, then they can always transition back to the same configuration. Using Lemma 21 and the fact that a final configuration cannot take any micro-step, Theorem 16 can be proved.

With these two theorems, we can always pick an implementation that will not get stuck or aborts abnormally for a well-formed monitor and all implementations generate the same verdicts as long as their behaviors follow the semantic rules. In the next section, we will use Fiat to generate an implementation by refinement.

3.1.4. Code generation by refinement using Fiat

Overview of Fiat. Stepwise refinement derives executable programs from nondeterministic specifications. In each step, some details of the computation are decided upon, proceeding this way until a computable program is derived. Each refinement step must not introduce new behavior: the values that a refined program may produce must be a subset of the values allowed by the specification. Fiat is a stepwise refinement framework, providing a semi-automatic way of deriving correct and efficient programs. Here *semi* means that while the derivation process is automatic, it depends on manually verified refinement lemmas, specific to the domain that Fiat is applied to. Readers can refer to [47, 36, 133] for more information.

Key syntax structures in Fiat. In Fiat, specifications are logical predicates characterizing allowable output values. These specifications are called computations, and written in the non-determinism monad: deterministic programs can be lifted into computations using the *ret* combinator, computations can be sequenced using the *bind* combinator (written "$x \leftarrow c_1; c_2(x)$"), and a nondeterministic choice operator written $\{a|P\ a\}$ is used to describe programs that may return any value satisfying a logical predicate P. Concretely, the result of binding two computations c_1 and c_2 as shown above is simply the set $\{y|\exists x, x \in c_1 \wedge y \in c_2(x)\}$.

Fiat computations are organized into an ADT (abstract data type), a structure used to encapsulate a set of operations on an internal data type. In Fiat, an ADT contains an internal representation type *rep*, a list of constructors for creating values of *rep*, and a list of methods operating on values of *rep*. A well-typed ADT guarantees that *rep* is opaque to client programs using the operations of the ADT.

Refinement calculus in Fiat. Refinement in Fiat is the process of transforming an ADT into a more deterministic one, involving refining all constructors and methods defined in it and picking an efficient internal representation using data refinement [76] of *rep*. When refining an expression, a partial relation $c_1 \supseteq c_2$ must be preserved for each refinement step, meaning that the possible values of expression c_2 must be a subset of the possible values of expression c_1. For the data refinement, changes of internal representation are justified using a user-selected abstraction relation, so that if the internal states of two ADTs are related, calling their methods must preserve the relation and produce the same client-visible outputs. Adding the abstraction relation r to the partial relation $\supseteq$ of refinement on expression, Fiat uses $\supseteq_r$ to represent the relation to be preserved for each refinement step: $I_1 \supseteq_r I_2 \supseteq_r ... \supseteq_r I_i$ where I_1 is the initial ADT and I_i is a *fully refined* (i.e. deterministic) ADT.

Figure 4 gives an overview of the code generation process. The initial ADT describes the basic behavior of monitors using semantic rules defined in the previous section. Then, the ADT is refined by proving a *sharpening* theorem, wherein the representation type, constructors and methods of the ADT are refined. The refinement of methods involves picking a specific implementation and

proving that $\supseteq_r$ is preserved between the specification and the implementation. The implementation is parameterized by a specific monitor definition given a starting state (configuration) and proof of well-formedness of that monitor. Executable code such as OCaml or Haskell can then be extracted from this definition.

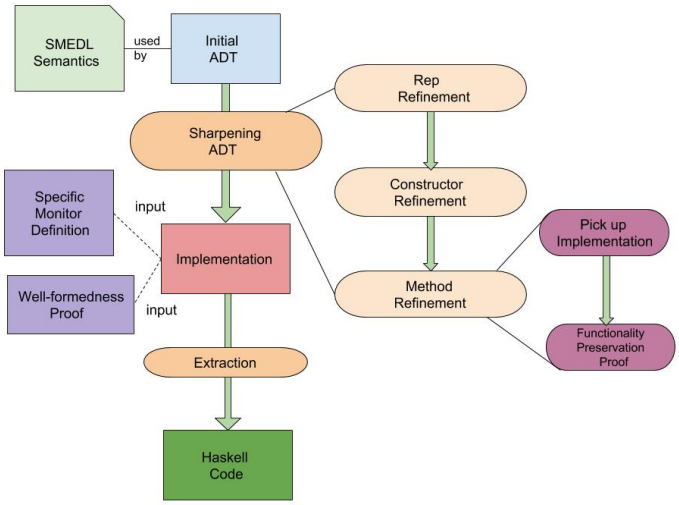

Figure 4: Code generation process

Definition of an ADT. The monitor ADT describes the common process of handling imported events using the semantic rules defined in the previous section. The definition of this ADT is given below.[2]

```
Definition confSpec : ADT _ := Def ADT {
  rep := configuration M,
  Def Constructor0 newS : rep := { c : configuration M | readyConfig c },,
  Def Method1 processEventS (r : rep) (e: raisedEvent | raisedAsImported  M e) :
      rep * list raisedEvent :=
      { p : rep * list raisedEvent
    | exists conf' econf,
        chainMergeTrans r conf' econf (` e) (fst p) (snd p) }
}.
```

The configuration of a given monitor *M* is used as the representation type for the ADT. Instead of constructing a concrete value, Constructor *newS* specifies that the starting state of a monitor should

[2]Some notations such as *Def*, *Constructor* and *Method*, are defined in Fiat

be a *ready* configuration. A ready configuration has empty sets for *PD* and *SC*, indicating that the monitor is ready to receive an imported event for the next macro-step. The method *processEventS* specifies the non-deterministic action of taking a macro-step. The first parameter *r* represents the current ready state of the monitor and the second parameter *e* is the imported event triggering the macro-step. The return value is a tuple of a ready configuration that reflects the updated state of the monitor after the macro-step and a list of raised exported events. The semantic rules from previous sections were defined in a relational way to conveniently specify this method, since relations easily model non-deterministic functions. To adapt the chain merge rule to the interface of *processEventS*, *chainMergeTrans* is defined below:

```
Definition chainMergeTrans {M : monitor} (conf conf' econf: configuration M)
(e: raisedEvent) (rconf: configuration M)  (events: list raisedEvent)  : Prop :=
   configTrans conf conf' /\
   chainMerge conf' econf e /\
   finalConfig econf  /\
   configTransRev econf rconf /\
   events = EX econf.
```

configTrans conf conf' represents the transformation from the ready configuration *conf* to initial configuration *conf'*; *chainMerge conf' econf e* is the Coq definition of $conf' \overset{e}{\rightharpoonup} econf$ with the number of steps taken omitted; and *configTransRev* represents the transformation from *econf* to a new ready configuration *rconf*. *events* is the set of exported events raised in this macro-step.

Refinement process. Refinement using Fiat requires proving the theorem $FullySharpened$ $(confSpec\ M)$, parameterized over some monitor definition *M*. The implementation is wrapped in the proof object of the theorem. The first step refines the representation type. Here we choose the same representation type—the configuration of monitor *M*—in the implementation. As a result, the *abstraction relation r* is plain equality. Constructor *newS* is refined by choosing a ready configuration *conf* for monitor *M*, given by the starting state of monitor *M*. Just like parameter *M*, *conf* also needs to be provided to generate a concrete, executable monitor. To refine method *processEventS*, we need to provide a deterministic function that preserves the semantics of applying the chain merge rule. Preservation of the specification's semantics for this function is given by the lemma below:

```
Lemma ProcessEventRefined M (C : configuration M) (W : Wellformed M)
      (Cor:readyConfig C) (e: raisedEvent) (P : raisedAsImported M e) :
 refine { p : configuration M * list raisedEvent
        | exists conf' econf, chainMergeTrans C conf' econf e (fst p) (snd p) }
        (ret (macroStepReadyFinal W Cor P (length (S M)))).
```

macroStepReadyFinal is a function which takes a ready configuration *C* and returns a new ready configuration and list of exported events. Here we choose a straightforward implementation: a fixpoint function that picks the first event from *PD* of the current configuration to trigger the next micro-step. Note that in the Coq definition, we use a list to represent the set, and due to the predicates establishing well-formedness, *PD* can never have duplicate events. The number of times the semantic function gets invoked is bounded by the number of scenarios in *M*. Provided that *M* is well-formed, it is guaranteed that the resulting configuration is a final configuration. The lemma *ProcessEventRefined* establishes that the return value is a subset of the results obtained by applying *chainMergeTrans* used in the original ADT. From the proof object of the theorem, an executable version of *processEventS* can be obtained.

It is worth noting that, the semantics of SMEDL can be directly expressed as a Coq function for generating the Haskell code by native Coq. But through Fiat, we can refine from the SMEDL semantics in relational style into a more efficient implementation by changing the data structure for configuration, handling pending events more wisely, etc. Moreover, refinement can be conducted in a more mechanical and extensible way in Fiat than using native Coq.

3.1.5. Case Study

A general event processing function is generated by refinement, parameterized by: a specific monitor specification, its well-formedness proof and a starting, ready state for that monitor. Therefore, to obtain a correct-by-construction monitor, one needs to 1) write a monitor definition *M*; 2) prove that *M* is well-formed; and 3) specify a starting state. A Haskell program is then be extracted, from which a monitor is implemented by adding glue code to receive events from the target system. This section uses a real-world monitoring requirement to illustrate the usability of this method.

SMEDL specification. The monitoring requirement comes from a known vulnerability (CVE-2017-9228),[3] in Oniguruma v6.2.0 [108] which is related to incorrect parsing of *character classes* in regular expressions, resulting in a crash due to access of an uninitialized variable. A character class is a pattern that represents a set of characters that is matched by a single character in the stream, if and only if that character belongs to the set. To parse a character set such as "[0-5]", a state machine is implemented, as shown in Figure 5. Omitted from the figure, transitions in the state machine are triggered by tokens read by the parser and guarded with addition conditions. For instance, the transition to the RANGE state requires a look-ahead token to recognize the '-' character.

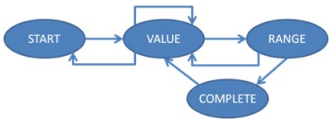

Figure 5: State machine of parsing character class

Based on the parsing state machine, and agnostic of the specific vulnerability, a SMEDL monitor is constructed. Part of the SMEDL specification (denoted as *parseCC*) is given below. We concentrate on a policy which says that the VALUE state cannot be reached from the START state when we expect to parse a character class next. There are two scenarios in *parseCC*. The scenario *main* models the transitions of the parsing state machine. The scenario *check_class* tracks whether to parse a POSIX character class next. Imported events starting with *state_to_* represent the transitions. The imported event *inClass* and *outClass* represent beginning and exiting of setting the next state to handle character classes. The monitor can detect the violation of this policy by raising the event *error* when the state of *main* transitions to VALUE while the state variable *in_class* is equal to 1. We encode this specification in Coq, which will then be generated into executable code.

```
object parseCC
    state:
        int in_class = 0;
    events:
        imported inClass();//enter next_state_class
        imported outClass();//exit next_state_class
```

[3]https://nvd.nist.gov/vuln/detail/CVE-2017-9228

```
        imported state_to_start();//state is set to START
        imported state_to_value();//state is set to VALUE
        imported state_to_range();//state is set to RANGE
        imported state_to_complete();//state is set to COMPLETE
        exported error(int);
    scenarios:
       main:
           START -> state_to_value() when (in_class != 1) -> VALUE;
           START -> state_to_value() when (in_class == 1)
           {raise error(0);} -> START;
           VALUE -> state_to_value() -> VALUE;
           VALUE -> state_to_range() -> RANGE;
           VALUE -> state_to_start() -> START;
           ...

       check_class:
          idle -> inClass() when (in_class == 0)
                {in_class = 1;} -> idle;
          idle -> outClass() when (in_class == 1)
                {in_class = 0;} -> idle;
```

Proof of well-formedness. Proving the well-formedness of a monitor seems hard because there are multiple sub predicates needed to be proved and type correctness needs to be checked. However, we have implemented decision procedures to check whether a monitor satisfies *P1* to *P3*. Rest of them can be proved using the auxiliary lemmas and tactics. The LOC of the proof is less than 1K of Gallina and Ltac code, which takes about 30 minutes to finish.

Construction of the Haskell monitor. The core building block of a *parseCC*-based monitor is given below. *processEventS* is the general event handling function refined from the Fiat ADT. The Parameter *r* contains the information to be used by *parseCC*: the proof of well-formedness (denoted as *Well_ParseCC*) and a starting state. Parameter *e* is the imported, triggering event for *parseCC*.

```
Definition parseCC_processEvent (r : ComputationalADT.cRep
            program Well_ParseCC configuration1_ready)
           (e: raisedEvent | raisedAsImported  parseCC e) :=
  processEventS r e.
```

After a Haskell program is extracted, the monitor is constructed by adding glue code for receiving events from the target system. We compile the Haskell code into an object file and expose two functions to be instrumented into the target program. The type signature of these two Haskell functions are given below:

```
cInitialRep :: IO (Ptr ())
cHandleImported :: CString -> Ptr () -> IO (Ptr ())
```

Both functions rely on the extracted Haskell code. *cInitialRep* provides a starting state for the monitor. *chandleImported* takes the name of an imported event, and the current state of the monitor, and returns a new state with any exported events printed out. The target system is responsible for recording this state update transparently. Using the Haskell compiler, both an object file and a C header file are generated. The header file contains the C API of the two functions, which are called in the source code of Oniguruma. When an incorrect transition occurs in the library, an alarm is raised and printed to the screen.

The difficult part of deriving a monitor is its proof of well-formedness, which can be simplified using the provided decision procedures and tactics. Other steps are easily implemented using common procedures. The methodology presented here provides an straightforward way to implement correct-by-construction monitors. One concern of using formal techniques is the manual effort involved in proof work. In our development, proofs are divided into two parts: one part includes proofs used during the refinement process, and auxiliary tactics and decision procedures for proving the well-formedness of any monitor; the other part is the proof of well-formedness for a particular monitor. The raw LOC in Coq for the first part of proof (excluding the Fiat code) is about 30k lines. However, to apply the technique, users only need prove well-formedness of their particular monitor, which is not labor-intensive given the help of auxiliary tactics and lemmas. Therefore, we assert that generating correct runtime monitors using a proof assistant is a feasible task.

3.2. SMEDL monitoring system

In the previous section, we have proposed SMEDL to describe properties in a single monitor. Complicated requirements can be specified in a modular way as communicating scenarios, which are convenient to design and organize. However, as software systems and properties become increasingly complicated, it is still intractable to solely use single monitors to fulfill all monitoring requirements. For instance, a property of a distributed system requires analyzing behaviors of sub components but observations from different components are independent with each other. In this case, rather than gathering all observations using a single monitor, it is more sensible to use monitors to check sub components locally and then aggregate verdicts in a downstream monitor. Moreover, target systems may evolve during runtime by extending dynamic data structure instances, adding or removing components. As presented in Chapter 4, many properties for such kind of system rely not only on the event order by also on the attribute values of which the domain are expanded during runtime. To handle them, it is desirable to equip monitors with ability to dynamically scale together with the target system. In this section, we propose SMEDL monitoring system as a monitor network. The SMEDL specification language is extended to express monitoring logic as a collection of monitors and monitoring architecture as flows of information between the target system and monitors. The system supports synchronous as well as asynchronous deployment of monitors and dynamic instantiation of monitors on demand. We will first introduce the overall system design and salient features. Then, we will define the architecture description and present a semantics of synchronous monitor network.

3.2.1. System design

Modular property specification. In order to effectively monitor a property in a large-scale distributed system, SMEDL allows us to specify properties in a modular fashion. In this way, a complex property can be decomposed into a set of monitoring modules that collectively implement the monitor for the overall property. A common pattern for modular specification is partitioning a global property for a distributed system into a set of locally deployed modules that operate on local observations of each process in the distributed system and convey results of local processing to the

global module that computes the overall result.

Monitor coordination and communication. After a complicated monitoring requirement is decomposed into multiple monitors, coordination between monitors needs to be achieved by communication. The flow of interactions between monitors depends on the property and how it is partitioned into modules. Specification of the monitoring architecture, described below, makes these flows explicit.

Synchronous and asynchronous deployment. We specify the logic of each monitor module and, separately, how this module is to be deployed. Often, the user has a choice of deploying the same module synchronously or asynchronously, so decoupling the logic of the module from its deployment strategy increases flexibility of the framework. Deployment of monitors also influences how monitors communicate with each other.

Dynamic monitor instantiation. Large-scale software systems typically contain many similar components that can be added and removed dynamically. Monitors can be instantiated according to the scaling of the target system.

Separation of property specification from observation extraction. A monitor specification describes, among other things, what observations are needed by the monitor in order to do its job. In order to deploy monitors, we also need to know how to extract these observations from the target system. Extraction of observations can be performed in many different ways, for example by instrumenting source code or binaries of system components, by snooping on the system bus, or even offline, reading from a recorded trace. Over time, the target system may evolve and offer new ways of observation extraction, or different variants of system component implementations may require different placement of instrumentation probes. It is important to accommodate these changes in the monitoring setup with as little disruption as possible. SMEDL separates monitoring logic from observation extraction using an event-based API, so that events can be raised in a specified format by an appropriate extraction technology. In our work, we have experimented with several such technologies, such as AspectC [37] for instrumenting C source code and a dynamic translation

tool for capturing observations from binary code. The extracted event can either be delivered to a synchronous monitor by calling an monitor API or to an asynchronous monitor by communication middleware such as RabbitMQ [116].

3.2.2. Monitoring architecture

Before introducing the monitor architecture, we first extend the definition of a monitor with a name and identity parameters (also referred to as parameter variables or just parameters). Choosing different parameter values allow us to have multiple *monitor instances*. To distinguish between a monitor and its instance, we also use the terminology *monitor type* to refer a monitor. To describe communication between monitors while hiding the internal structure of the monitoring logic, we define the interface for a monitor.

Definition 22 (Monitor interface) *A monitor interface is a triple* $\langle monName, paras, EvInterface \rangle$ *where* $monName$ *is the name of the monitor;* $paras$ *is a list of typed parameter variables;* $EvInterface$ *is a list of imported and exported events of the monitor.*

Monitoring architecture is a directed graph that represents communication between monitors (or monitors and the target system) that involve in a monitoring requirement. Nodes of the graph have ports that correspond to events that the node can consume or produce. Ports of monitor nodes match the interface of the monitor. Ports of the target system node represent observations that are obtained from it. Edges in the graph represent communication flows from exported events of one node to imported events of another node. Nodes in a monitoring architecture are partitioned into *synchronous sets*. Monitors within a synchronous set use a single thread of control while communication between synchronous sets is asynchronous. During runtime, a monitor instance may be created either statically at the beginning of a monitored run of the system or dynamically when new values of parameters are discovered. The formal definition is given below.

Definition 23 (Architecture description) *An architecture description is a triple* $\langle MonDef, Sync\ Def, Channel \rangle$ *where* $MonDef$ *is a set of monitor interfaces;* $SyncDef$ *partitions monitors in* $MonDef$ *into synchronous sets; and* $Channel$ *is a set of* $event\ connection\ specifications$.

An event connection specification is a tuple ⟨*SrcMon, SrcEv, TarMon, TarEv, MonArgs, EvArgs*⟩, which specifies how an exported event *SrcEv* of a source monitor (or the target system) *SrcMon* is delivered to a target monitor *TarMon* as its an imported event *TarEv*. *MonArgs* (*EvArgs*) is a set of *PatternExpr* specifying how to bind a parameter variable in the target monitor or an attribute of the target event with a value from the source monitor or the source event. Each parameter of a monitor or an attribute of an event corresponds to an index according to its position in the parameter or attribute list, starting from 0. A *PatternExpr* is a tuple ⟨*targetIdx, source, sourceIdx*⟩, meaning that the parameter value of *TarMon* (in *MonArgs*) or the attribute value of *TarEv* (in *EvArgs*) with index *targetIdx* must be matched to the parameter value of *source* with index *sourceIdx*. *source* can be either *SrcMon* or *SrcEv*. To correctly bind values, the type information of parameters and event attributes carried by *MonDef* are used. Moreover, all attributes in *EvArgs* must be assigned with a value. If all parameters are assigned with a value for *MonArgs*, the connection is *unicast* because at most one instance can be the recipient. For multicast actions, one or more positions can be assigned with an wildcard expression so that multiple instances may receive the event. For a unicast connection, *TarEv* is also called an (implicit) *creation* event of *TarMon* because it is used to create an instance when there is no instance that can consume the incoming event.

For example, an event connection specification $(mon1, e1(x, y), mon2, e2(z), MonArgs : \{(0, mon1, 0), (1, e1, 0)\}, EvArgs : \{(0, e1, 1)\})$ specifies that instances of $mon1$ send $e1$ to $mon2$ as $e2$. Note that type information of monitors and events are omitted here. When an event instance $e1(x1, y1)$ is sent from an monitor instance $mon1(a1)$, the monitor instance $mon2(a1, x1)$ receives it as $e2(y1)$ as the specification requires that the first and second parameter of $mon2$ respectively match to the first parameter of $mon1$ and the first attribute of $e1$ and the attribute of $e2$ matches to the second attribute of $e1$. If there is no instance of *mon2* parameterized with $(a1, x1)$, $mon2(a1, x1)$ will be created. The concrete syntax of the architecture description is given in Appendix A.1.2.

3.2.3. Semantics of synchronous set

Monitors in a monitor network are partitioned into synchronous sets in *SyncDef*. Monitors belonging to the same set communicate with each other synchronously. This section formally defines the semantics for a synchronous set. We first define a monitor instance as a tuple $\langle$*monName, parasBinding*$\rangle$ where *monName* is the name of the monitor and *parasBinding* is a map from *paras* to values they are bound to. For a *raised event*, we extend its definition with *monInst* to denote the monitor instance that raises it. The dynamic state of an instance is still represented by a configuration and the configuration of a monitor M is a partial function $monConfig_M$ from instances of M to configurations. The configuration of a synchronous set $MSet$ is defined as $setConfig_{MSet} = \{monConfig_M | M \in MSet\}$.

With our definitions in place, we now describe the semantics in three parts: first, we define the *local rules* for handling an event for a specific a monitor. Since an event may be delivered to multiple monitor types, we define the *set rules* for handling an event for a set of monitors. We finally define the *global rules* to coordinate execution of an entire synchronous set. Each semantic rule takes the architecture description $arch$ as an implicit parameter.

Local rules. The local rules describes the behavior of a monitor M handling an event *ei*.

$$\frac{\begin{array}{c} getInfo(monConfig_M, ei, arch) = (ids, type, e) \\ monConfig_M \downarrow^{ids}_{e,type} (monConfig'_M, l) \end{array}}{monConfig_M \nearrow_{ei} (monConfig'_M, l)} \text{[local-step]}$$

The function $getInfo(M, ei, arch)$ retrieves instances of M that consume *ei* by looking up the appropriate event connection definition in *arch*. *ei* carries the information of the *SrcMon* and *SrcEv* while M is the target monitor *TarMon*. All matched instances of M are stored in *ids*. Note that if there is no corresponding instance, *ids* represents parameters of the instance to be created. *ei* is mapped to the raised event e to be consumed by *ids*. Moreover, by checking *MonArgs* of the matched connection, we can also know whether it is *unicast* or *multicast*. Stored in *type*, we will use it to apply different semantic rules below.

The second portion of the rule above the line is defined in terms of the relation $\downarrow^{ids}_{e,type}$, which we define in terms of the action type we will perform. For presentation purposes, we will condense the type arguments to *U* and *M*, corresponding to unicast and multicast respectively.

Unicast actions. The semantic rules for a unicast connection are defined in two cases. If the instance already exists, unicast events proceed as:

$$\frac{\begin{array}{c} ids \in dom(monConfig_M) \\ monConfig_M(ids) = conf \\ conf \xrightarrow{e} conf' \end{array}}{monConfig_M \downarrow^{\{ids\}}_{e,U} (monConfig_M[ids \mapsto conf'], \{(e', ids) | e' \in conf'.EX\})} \text{[unicast-no-create]}$$

The condition guarantees that the instance ids already exists and the corresponding configuration is $conf$. By taking a macro-step, the state of the instance ids is updated to $conf'$. The raised exported events are extended with ids.

If the monitor instance does not exist, we must first create an instance with ids before performing the macro-step, as shown in the rule below.

$$\frac{\begin{array}{c} ids \notin dom(monConfig_M) \\ initDefaultConf(ids) = conf \\ conf \xrightarrow{e} conf' \end{array}}{monConfig_M \downarrow^{\{ids\}}_{e,U} (monConfig_M[ids \mapsto conf'], \{(e', ids) | e' \in conf'.EX\})} \text{[unicast-create]}$$

Finally, as part of the reduction, we update *monConfig* with this resultant instance and return any events that macro-step produced.

Multicast actions. Multicast connections may need to update a number of objects over the course of reduction. To this end, we define multicast operations. The first sub-rule addresses the case where there is no instance.

$$\frac{}{monConfig_M \downarrow^{\emptyset}_{e,M} (monConfig_M, \emptyset)} \text{[multicast-base]}$$

The second case is defined by using the unicast rule on each single instance. Note that the final union $I_1 \cup I_2$ will be a disjoint union as each entry in I is parameterized by the ids used to create it.

$$\frac{monConfig_M \downarrow_{e,U}^{\{ids\}} (newConfig_M, I_1) \quad newConfig_M \downarrow_{e,M}^{rest} (finalConfig_M, I_2)}{monConfig_M \downarrow_{e,M}^{\{ids\} \cup rest} (finalConfig_M, I_1 \cup l_2)} \text{[multicast-big-step]}$$

Set rules. Multiple monitors may handle an incoming event. To achieve parametric monitoring presented in Chapter 4, we need to specify an order between monitors. The order is decided by relation of parameters between monitors. For two monitors, m_1 and m_2, if a parameter variable p of m_1 matches with a parameter value q of m_2 in an event connection specification regardless of whether m_1 and m_2 are the source of the target monitor, we say p *relates* to q. Here we require that the relation of parameters for two monitors are fixed, which means if p of m_1 relates to q of m_2, there is no other parameters in m_2 (m_1) that matches with p (q). The partial order ($\leq$) between monitors are defined upon relation of parameters between two monitors. For two sets of parameters θ_1 and θ_2 for respectively for m_1 and m_2, if all parameters of θ_1 are related to parameters in θ_2, we have $m_1 \leq m_2$. Then, we can order monitors in the synchronous set *MSet* in the $monTypeList_{MSet}$ according to this partial relation: if $m_1 \leq m_2$, the index of m_2 is less than m_1 in it. During execution, if an event can be handled by multiple monitors, the monitor with smaller index in *monTypeList* will handle this event before ones that with a larger index.

To this end, we define set rules as reduction relations over set configurations as follows. We first inductively define $\Rightarrow_{MSet}^{MLst}$, the basic set configuration reduction relation for $MLst$, which is a sub list of $monTypeList_{MSet}$.

The set of raised events L is divided into two parts L_1 and L_2. L_1 is the set of events to be consumed by monitors in $MSet$ while L_2 are events to be sent to other synchronous sets. Note that the dot operator appends two lists and $MSet.monName$ denotes all names of the monitors in the set $MSet$.

$$\frac{\begin{array}{c} monConfig_M \nearrow_{ei} (monConfig'_M, L) \\ L_2 = L \setminus L_1 \quad L_1 = \{eo | eo \in L \wedge \exists ch, ch \in arch.Channel \wedge \\ ch.SrcMon \in MSet.monName \wedge ch.SrcEv = eo.event.evName\} \\ setConfig' \overset{ei}{\Rightarrow}{}^{monTypeList'}_{MSet} (setConfig'', L'_1, L'_2) \end{array}}{\begin{array}{c} setConfig' \cup \{monConfig_M\} \overset{ei}{\Rightarrow}{}^{M::monTypeList'}_{MSet} \\ (setConfig'' \cup \{monConfig'_M\}, L_1.L'_1, L_2 \cup L'_2) \end{array}} \text{[set-reduce-inductive]}$$

$$\frac{}{setConfig \overset{ei}{\Rightarrow}{}^{[]}_{MSet} (setConfig, nil, \emptyset)} \text{[set-reduce-base]}$$

The sub rule *set-reduce* states that the execution of the set rule starts by picking up monitors in the synchronous set $MSet$ that can consume ei. The function $sort$ is to sort monitors in $MSet$ according to the partial order $\leq$.

$$\frac{\begin{array}{c} ML = \{M | M \in MSet \wedge \exists ch, ch \in arch.Channel \wedge \\ ch.SrcEv = ei.event.evName \wedge ch.TarMon = M.monName \\ \wedge ch.SrcMon = ei.monInst.monName\} \\ monTypeList_{MSet} = sort(ML) \\ setConfig \overset{ei}{\Rightarrow}{}^{monTypeList_{MSet}}_{MSet} (setConfig', L_1, L_2) \end{array}}{setConfig \overset{ei}{\Rightarrow}_{MSet} (setConfig', L_1, L_2)} \text{[set-reduce]}$$

Global rules. The global rule for the synchronous set $MSet$ is defined as the update of configurations carrying two event collections: one to be consumed within the synchronous set and the other one to be sent to the environment:

$$(setConfig, evQueue, exSet) \Downarrow_{MSet} (setConfig', evQueue', exSet')$$

$setConfig$ and $setConfig'$ are configurations of $MSet$ before and after applying our global rules; $evQueue$ and $evQueue'$ are the queue of events to be consumed within the monitor before and after a step; and $exSet$ and $exSet'$ are the sets of events to be sent to the environment. Note that each element of $evQueue$ is a set of events, which is obtained when applying the local rule. This relation has two reduction rules indicating how to retrieve the next event instance to process and how to

remove an empty event instance set.

The first rule represents fetching a single event from the head of the queue, triggering its execution and adding any resultant external events to the back of the external set:

$$\frac{setConfig \overset{e}{\Rightarrow}_{MSet} (setConfig', L'_1, L'_2)}{(setConfig, (\{e\} \cup C) :: L_1, L_2) \Downarrow_{MSet} (setConfig', C :: L_1.L'_1, L'_2 \cup L_2)}$$

We can observe that this rule leaves flexibility on which event to fetch next. As a future work, we would adopt the similar method to define predicates on the structure of a synchronous set to guarantee determinism. The second rule specifies how to proceed when the first event set in the incoming event queue is empty, moving down the list of events:

$$\frac{}{(setConfig, (\{\}) :: L_1, L_2) \Downarrow_{MSet} (setConfig, L_1, L_2)}$$

Note that nil is a valid list, and so $\{\} :: nil$ will reduce to nil, cleaning up the remaining empty set in the queue. If $evQueue$ is empty, events in $exQueue$ will be sent to the environment. Then, another event from the environment can be sent to $evQueue$. The overall top level rule for the evolution of synchronous set is then defined as below. With this rule, we can abstract each synchronous set as an atomic entity when defining semantics of a monitor network with multiple synchronous sets.

$$\frac{(setConfig, \{e\} :: nil, \emptyset) \Downarrow_{MSet} (setConfig', \{\}, L)}{setConfig \Downarrow^{e}_{MSet} (setConfig', L)} \text{[synchronous-set-evolution]}$$

3.2.4. Case Study

To further motivate the design of SMEDL and illustrate its usability, we apply SMEDL to *Case 2* in Section 2.1. More examples can be found in Appendix A.2 and corresponding explanations of them will be given in Chapter 5. Consider the design of a track duration monitor. Recall that a track is observed as a sequence of timestamped points. Each new point added to the track results in a *track* event. The monitor consumes the track events and calculates the average duration of tracks over sliding windows. Here we use SMEDL to specify this requirement. Compared to high-level

languages, monitoring logic can be expressed described by communicating monitors. Concerns of computation and sliding window can be separated. Compared to general framework, using well-designed DSL may also reduce the chance to make mistakes.

For the concrete design, because each track event carries the track identifier, there is a local monitor for each track that calculates duration of the track in the current window and, at each window boundary, sends the value to the global monitor to calculate the metric for all tracks. As tracks are added by the application, new track monitors are instantiated. To implement calculation of track duration over a sliding window, the window is partitioned into a series of sub windows, each represented by a separate monitor. A window manager monitor for each track handles moving of sub windows, while the aggregator monitor combines calculations from each sub window into the overall track duration within the whole window. The architecture of the monitor is shown in Figure 6 (a). Some events are not shown for clarity, including ones that are sent from the environment to trigger execution of the monitor network. Each box represents a monitor, with types of monitor parameters shown in brackets. Edges represent events exchanged by monitors. Each edge is annotated with parameter matching that determines replication of event flows when new instances are created. Consider, for example, the *track* event raised by *WindowManager* and consumed by *Subwindow*. The matching ties the first parameter of the *WindowManager* instance raising the event to the first parameter of the *Subwindow* instance receiving the event. Since *Subwindow* has the second parameter, not bound by the matching, the connection is a fan-out, when the track event is received by all instances of *Subwindow* for that track. By contrast, event *metric_sub* represents a fan-in, when events raised by any sub window for a track are delivered to the instance of *Aggregator* for that track. Finally, *metric* events raised by any instances of *Aggregator* are delivered to the same *Metric* monitor, which is not parameterized. The *add_track* event is sent to *Metric* whenever a new instance of *WindowManager* is created. An instance of the architecture for two tracks, and two sub windows in a window, is shown in Figure 6 (b).

We illustrate a single monitor specification using a simplified version of *Aggregator*, shown below. A number of state variables are defined. It has two imported events, one representing a report from

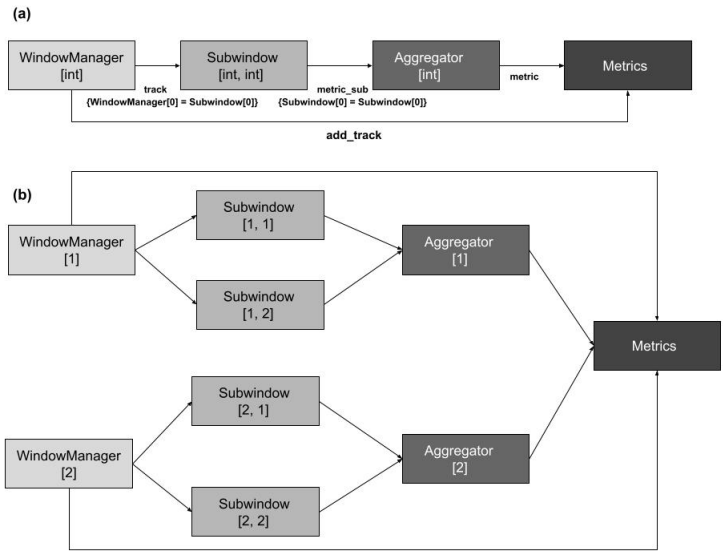

Figure 6: Monitoring architecture for the tracker monitor

a sub window and the other used for initialization, and one exported event, representing the track duration calculated at the window boundary. It also has a number of internal events, described below. Monitoring logic is represented by a collection of scenarios. In this example, each scenario has a single state. Each transition in a scenario is triggered by an imported or internal event and can happen only if a guard is satisfied. Guards are predicates over state variables of the monitor and attributes of the triggering events. When a transition occurs, a series of actions is executed, each of which either updates a state variable or raises an exported or internal event. For clarity, we do not show details of the guards and elide most of the actions. We can see that each scenario performs a certain check represented as a guard. For example, the check can determine whether the track started or was dropped within the current window, and updates the state variables accordingly. Then, an internal event is raised to trigger the next check.

On deployment of the monitor network. There are multiple ways of how four monitors coordinate with each other. If all of them are placed in the same synchronous set, the behavior is determined by the semantics of synchronous set proposed above. For instance, when a new track is observed, a new instance of *WindowManager* is created, which then sends an *add_track* event to *Metrics*. During this process, *WindowManager* will not receive any events from the environment until *Metrics* finishes its

execution. However, if *WindowManager* and *Metrics* are not in the same synchronous set, they can execute in parallel and there is no order guarantee between events delivered to a monitor instance. For instance, the *timeout* event (not shown in Figure 5) is consumed by *WindowManager* to move the sliding window. At the same time, it triggers the computation of metrics for each track as well. *Metrics* receives the verdict from *Aggregator* for each track and computes an average value of them. However, if a new track is observed and *Metrics* receives an *add_track* event before collecting all verdicts from existing aggregators, the final verdict will be incorrect because the number of tracks stored in *Metrics* reflects the new track. Whether this behavior happens depends on the underlying mechanism of asynchronous communication, which is out of the scope of this thesis.

```
object Aggregator
    state:
        int msg_cnt = 0;
        int event_cnt = 0;
        boolean g1, g2, g3;
        float observed_time = 0;
    events:
        imported initial(int, float, int, int);
        imported metric_sub(int, float, float, int);
        internal checkNum();
        internal i1(int, float, float);
        internal i2(float, float);
        internal i3(float);
        internal output();
        exported metric(int, float);
    scenarios:
       initialization:
           init -> initial(ts, sub_w, sub_size, prob) {...} -> init;
       accumulation:
           start -> metric_sub(n, ft, lt, flag) {msg_cnt = msg_cnt + 1; g1 = ...; raise i1(n, ft, lt); } -> start;
       chk_n:
           in -> i1(n, ft, lt) when (g1) {event_cnt = event_cnt + n; g2 = ..; raise i2(ft, lt); } -> in;
               else {event_cnt = event_cnt + n; raise checkNum();} -> in;
      check_ft:
         in -> i2(ft, lt) when (g2) {...; g3 = ..; raise i3(lt)} -> in;
                else {raise i3(lt)} -> in;
      check_lt:
        in -> i3(lt) when (g3) {...; raise checkNum();} -> in;
                else {raise checkNum();} -> in;
      check_num:
        in -> checkNum() {observed_time = ...; raise output();...;} -> in;
       output:
         in -> output() {raise metric(event_cnt, observed_time); ...;} -> in;
```

3.2.5. Discussion

The form of SMEDL monitor network is similar to general stream processing programs in the sense that single monitors transform the input events into output events and multiple monitors can coordinate with each other to achieve an overall monitor requirement by forming into a DAG. However, they are different in many aspects such as expressiveness, deployment of the system and the form of target programs, as shown in Table 2. From the perspective of expressiveness, stream processing frameworks support built-in transform operators such as sliding windows while users need to model operators as monitor specifications, like the case 2 illustrated above. Moreover, stream processing frameworks can handle large scale data with support of mature architecture and efficient implementation. However, SMEDL also has its advantages. SMEDL supports both synchronous and asynchronous deployment of monitors. Moreover, the implementation of a SMEDL monitor does not rely on heavy-weight runtime specifically designed for distributed systems. For instance, the monitor for *Case 2* can handle thousands of dynamically created tracks very efficiently without using multiple processes or threads. In Chapter 5, we will further demonstrate efficiency of SMEDL monitors.

Table 2: Comparison between stream processing and SMEDL

	Sstream processing framework	SMEDL
Expressiveness	predefined transform operators	transform operators defined manually
Deployment	asynchronous deployment	synchronous and asynchronous deployment
Target program	suitable for large scale systems	support for different sizes of programs

As stated in Section 2.2.6, SRV techniques are good at monitoring properties over data streams. For instance, QRE supports operations over streams at the language level while users need to manually encode them in SMEDL specifications. Moreover, the synthesis algorithm of QRE can generate efficient implementation from succinct specifications. However, using state machine as the formalism, SMEDL specifications are imperative and thus more intuitive, especially for users who are more familiar with general programming languages.

3.3. Summary

In this chapter, we proposed a specification language, SMEDL for runtime verification. Composed of a set of EFSMs, the single monitor specification can describe different forms of properties. To bridge the gap between the specification and the implementation, we proposed a method of correct-by-construction code generation for SMEDL monitors using the Fiat framework. We further extended SMEDL with an architecture description which specifies how single monitors communicate with each other to form a dynamically scalable monitor network. In the next chapter, we will present a method to use a monitor network to efficiently monitor parametric properties.

CHAPTER 4 : Parametric Monitoring Using SMEDL

Many real-world programs may scale in the sense of data or control, probably by creating new instances of data structure, thread or process. These instances are usually distinguished by identities. In the event-based RV, when events are extracted from operations of these instances, identity values are also attached to them as attributes. We denote these events as *parametric* events. Defined upon traces with parametric events, *parametric properties* not only depend on event order in the trace but also on the attribute values of events. As an example, the *UnsafeMapIter* [101] property shown below says an iterator of a collection created from a map is not allowed to be used after the map has been updated. The property is described as a regular expression and the alphabets of this property are parametric events: $createC(m, c)$ denotes creation of a collection c, the key set of a map m; $createI(c, i)$ is creation of iterator i from c; $updateM(m)$ is update of m; and $useI(i)$ is use of i.

Example 1 (UnsafeMapIter):

$$createC(m, c)updateM(m)^{*}createI(c, i)useI(i)^{*}updateM(m)^{+}useI(i)$$

When a monitor inspects event attributes during verification, we call it a *parametric monitor* which checks *parametric properties*. These properties can be monitored using multiple techniques [75] such as first-order temporal logic [18], monitoring modulo theories [46], rule-based system [114], stream processing [44] and trace slicing [5]. These techniques describe and check parametric properties in different ways and it is hard to directly compare them with respect to expressiveness and efficiency. In the previous section, we proposed the SMEDL framework, which utilizes dynamically scalable monitor networks to specify and check different types of properties for complicated software systems. In this chapter, we propose a common ground for parametric monitoring using SMEDL. More specifically, we focus on the trace slicing. The core idea of trace slicing is to slice a parametric event trace into sub traces according to event parameters. Each sub trace is constructed based on a binding from parameter variables to values and is checked against a non-parametric property. A trace slicing algorithm is implemented in MOP [101], which provides an efficient indexing mechanism to reduce monitoring overhead on both time and memory usage. QEA [14] adopts a

similar slicing mechanism but further supports aggregation of sub traces by explicit quantification of parameter variables. However, it requires full combinations of parameters across all values in their domains, which is different from MOP. Although it is in general more expressive, it is less efficient in memory and time overhead because more instances may be maintained.

We present a novel perspective of parametric trace slicing based on SMEDL. Intuitively, we propose a method to naturally describe parametric trace slicing and the aggregation semantics using a dynamically scalable monitor network. In this chapter, we :

- introduce preliminaries, including common notations and trace slicing monitoring of MOP and QEA. (Section 4.1)
- present a method to encode the trace slicing semantics by transforming from MOP to SMEDL with a formal proof of correctness. (Section 4.2)
- propose two syntactic fragments of QEA that can be transformed into equivalent SMEDL monitors and by applying the MOP slicing algorithm encoded in SMEDL, QEA properties can be efficiently monitored. (Section 4.3)

4.1. Preliminaries

4.1.1. Common definitions and notations.

Although definitions and notations related to trace slicing have been defined in MOP [34] and QEA [14], for self-containment and unification of terminologies, some common notations in both MOP and QEA are given in Table 3.

The relation between a parametric event type $e(\bar{x})$ and the ground event $e(\bar{v})$ is built by θ where θ is a partial function $X \rightharpoonup Val$ of which the domain $dom(\theta) = \bar{x}$ and $\theta(\bar{x}) = \bar{v}$. θ is also called as a $binding$ or a $parameter\ instance$. We will use the notation $e(\theta)$ to denote a parametric event from which the event type $e(\bar{x})$ and the ground event $e(\bar{v})$ can be obtained.

The following definitions are used to describe relations between bindings:

Variables	Var			Set of all variables
	X, W	$\subset$	Var	Parameter Sets
	x, w	$\in$	Var	
Values	Val			Set of all values
Events	Σ			$\Sigma \cap Var = \emptyset$
Trace Sets	Σ^*			
Parametric Event	$(e, \bar{x}), e(\bar{x})$			$e \in \Sigma, \bar{x} \subset Var$
Ground Event	$(e, \bar{v}), e(\bar{v})$			$e \in \Sigma, \bar{v} \subset Val$
Events	$\Sigma(X)$			$\{e(\bar{x}) \mid e \in \Sigma, \bar{x} \subseteq X\}$
	$\Sigma\langle X \rangle$			$\{e(\bar{v}) \mid e(\bar{x}) \in \Sigma(X), ground\}$
Traces	τ	$\equiv$	$\overline{e(\bar{v})}$	

Table 3: Common notations

Definition 24 (Containment) *If $dom(\theta_1) \subseteq (\subset)\ dom(\theta_2)$ and $\theta_1(x) = \theta_2(x)$ for all $x \in dom(\theta_1)$, we say θ_1 has equal or less (proper less) information than θ_2, denoted as $\theta_1 \sqsubseteq (\sqsubset)\ \theta_2$.*

Definition 25 (Compatible) *Two bindings θ_1 and θ_2 are compatible with each other (denoted $\theta_1 \sim \theta_2$) when $\theta_1(x)$ is equal to $\theta_2(x)$ for all $x \in dom(\theta_1) \cap dom(\theta_2)$,*

Definition 26 (Consistent) *Given a set of bindings Θ, if for any two bindings $\theta_1 \in \Theta$ and $\theta_2 \in \Theta$, we have $\theta_1 \sim \theta_2$, then Θ is consistent.*

Definition 27 (Combination) *The combination between two bindings θ_1 and θ_2 is defined as follows: if $\theta_1 \sim \theta_2$, $\theta_1 \sqcup \theta_2(x) = \theta_1(x)$ 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. $\theta_1 \sqcup \theta_2$ is the least upper bound (lub) of θ_1 and θ_2. We can also lift the lub operator to a set of bindings [112]: $\sqcup\Theta \equiv \theta_1 \sqcup ... \sqcup \theta_k$ where $\Theta = \{\theta_1, ..., \theta_k\}$.*

Definition 28 (Lub-closed) *A set of bindings Θ is lub-closed iff for any $\Theta' \subseteq \Theta$, if Θ' is consistent, $\sqcup\Theta' \in \Theta$.*

We also define a predicate $max(\theta_1, \Theta, \theta)$ which says θ_1 is a maximal binding in Θ that has equal or less information than θ:

$$max(\theta_1, \Theta, \theta) \equiv \theta_1 \sqsubseteq \theta \wedge \theta_1 \in \Theta \wedge (\forall \theta_2, (\theta_2 \in \Theta \wedge \theta_2 \sqsubseteq \theta) \implies \theta_2 \sqsubseteq \theta_1)$$

4.1.2. Parametric trace slicing using MOP

In MOP, a slice is defined with respect to a parameter instance: given a parameter instance θ and a parametric trace $\tau \in \Sigma\langle X\rangle^*$, a θ-trace slice $\tau \restriction_\theta \in \Sigma^*$ is a *non-parametric* trace. Each non-parametric event $e \in \tau \restriction_\theta$ corresponds to a parametric event $e(\theta') \in \tau$ where $\theta' \sqsubseteq \theta$. According to the definition, all events of which parameters are not compatible with θ or have more information than θ will not be in the θ-trace slice. A parametric property is a function of which the domain is a parametric trace. It outputs a map from parameter instances to boolean verdicts, which is obtained by checking against the non-parametric property for each slice sub trace.

A MOP monitor $M(X)$ is a five-tuple $\langle St, \Sigma(X), \iota, \sigma : St \times \Sigma(X) \rightarrow St, F\rangle$ where X is a set of parameter variables, *St* is a set of states, $\Sigma(X)$ is a set of parametric events, ι is the initial state, σ is the transition function and F is a set of final states.[4] MOP achieves parametric slicing using an algorithm $\mathbb{C}\langle X\rangle$ [34], presented as Algorithm 1 below. Parameterized with an parameter monitor $M(X)$, Algorithm 1 maintains and updates Δ and $\mathbb{U}$ reacting to the incoming parametric event $e(\theta)$. Δ stores states for parameter instances while $\mathbb{U}$ maps a parameter instance θ to all instances in the domain of Δ that have more information than θ. If θ is not defined in Δ, the algorithm adds θ to Δ by setting the state of the largest binding defined in Δ that is less informative than θ by traversing in the reverse topological order (line 7 - 10). If there is no such binding and e is a creation event, θ is added to Δ by assigning the initial state ι to it.

After θ has been added to Δ, it will be used to create bindings by extending the existing compatible bindings in Δ (line 13 - 16). Finally, e updates $\Delta(\theta)$ and all instances that are more informative than θ (line 17). We can prove that the domain of Δ is lub-closed, which will be used below when proving the relation between MOP and SMEDL.

Lemma 29 *$dom(\Delta)$ is lub-closed.*

Proof(sketch): the proof is performed by induction on the length of the input trace τ. The basic step is straightforward since the domain of Δ only has one element after receiving the event $e(\theta)$. In

[4] We modify the original definition of M(X) in [101] by adding events with parameter variables.

Algorithm 1 $\mathbb{C}\langle X\rangle(M = \langle St, \Sigma(X), \iota, \sigma, \gamma\rangle)$

```
1:  mapping Δ : [[X ⇀ Val] ⇀ St]
2:  mapping 𝕌 : [[X ⇀ Val] → P_f([X ⇀ Val])]
3:  Δ(θ) ← undefined for any θ ∈ [X ⇀ Val]
4:  𝕌(θ) ← ∅ for any θ ∈ [X ⇀ Val]
5:  procedure MAIN(e(θ))
6:      if Δ(θ) undefined then
7:          for θ_m ⊏ θ (in reversed topological order) do
8:              if Δ(θ_m) defined then goto 9
9:          if Δ(θ_m) defined then
10:             defineTo(θ,θ_m)
11:         else if e is a creation event then
12:             defineNew(θ)
13:         for θ_m ⊏ θ (in reversed topological order) do
14:             for θ_comp ∈ 𝕌(θ_m) compatible with θ do
15:                 if Δ(θ_comp ⊔ θ) undefined then
16:                     defineTo(θ_comp ⊔ θ,θ_comp)
17:     foreach θ' ∈ {θ} ∪ 𝕌(θ) do Δ(θ') ← σ(Δ(θ'), e) endfor
18: procedure DEFINENEW(θ)
19:     Δ(θ) ← ι
20:     for θ'' ⊏ θ do 𝕌(θ'') ← 𝕌(θ'') ∪ {θ}
21: procedure DEFINETO(θ, θ')
22:     Δ(θ) ← Δ(θ')
23:     for θ'' ⊏ θ do 𝕌(θ'') ← 𝕌(θ'') ∪ {θ}
```

the inductive step, suppose after consuming a trace τ, $dom(\Delta)$ is lub-closed. After receiving $e(\theta)$, if $\theta \in dom(\Delta)$, $dom(\Delta)$ is unchanged. Otherwise, given an arbitrary $\Theta' \subseteq dom(\Delta')$ where Δ' is updated from Δ by receiving $e(\theta)$, if there exists a set of bindings $\Theta'' \subseteq \Theta'$ and $\Theta'' \cap dom(\Delta) = \emptyset$, then there exists $\theta'' \in \Theta''$ is constructed by combining θ with an existing binding in $dom(\Delta)$. It is then easy to deduce that there exists $\Theta_1 \subseteq dom(\Delta)$ that $\sqcup\Theta' \equiv \theta \sqcup (\sqcup\Theta_1)$. According to Algorithm 1, $\mathbb{U}(\emptyset) = dom(\Delta)$ after handling τ. Therefore, θ will be combined with all compatible bindings in $dom(\Delta)$ and the result bindings are in $dom(\Delta')$. On the other hand, due to the inductive hypothesis, $\sqcup\Theta_1 \in dom(\Delta)$, so $\sqcup\Theta' \equiv \theta \sqcup (\sqcup\Theta_1) \in dom(\Delta')$. □

4.1.3. Parametric slicing using QEA

A QEA (quantified event automata) $Q(\Lambda)$ contains two parts. Q is an *event automaton* (EA) and $\Lambda \in (\{\forall, \exists\} \times vars(Q) \times Guard)^*$ is a list of quantifiers with guards of boolean expressions on

parameters. An EA is an EFSM in which transitions are enriched with guard and assignments to local variables; $vars(Q)$ is the set of parameter variables appearing in Q.

QEA uses trace slicing to accomplish parametric monitoring. The big-step semantics for acceptance for a parametric property for QEA is illustrated below [14]. $\theta_1 \dagger \theta_2$ overrides the value in θ_1 by θ_2; $g(\theta)$ is the guard condition over the quantified variable; $Q(\theta)$ is an event automaton Q with its variables instantiated by θ; $\tau \downarrow_{Q(\theta)}$ is the projection of a trace τ over $Q(\theta)$; $L(Q(\theta))$ is the set of traces accepted by $Q(\theta)$. Bindings are generated by inductively traversing the derived domain of each variable in the nested quantifiers. A *full binding* is a binding where all parameters in Λ are bound with values. The verdict is computed over verdicts of all created full bindings.

Definition 30 (Acceptance in QEA) *A QEA $Q(\Lambda)$ accepts a ground trace τ if $\tau \models_{\langle\rangle} \Lambda$ where $\models_\theta$ is defined as:*

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

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

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

4.2. Expressing trace slicing of MOP using SMEDL

This section presents how SMEDL expresses the trace slicing semantics by proposing a transformation from a MOP monitor to a SMEDL monitor network. We prove that the monitoring network in SMEDL is equivalent to Algorithm 1 with respect to trace slicing.

4.2.1. Transformation from MOP to SMEDL

Intuitively, MOP slices a trace of parametric events into sub traces, which is identified by a binding of parameter variables. By analyzing $M(X)$ statically, we can know which parameters will be combined together and how bindings are generated. Each possible combination of parameters corresponds to a single SMEDL monitor and the connection between two monitors represent the behavior of creating a new binding by extending an existing one. Consequently, $M(X)$ is transformed

into a set of SMEDL monitor specifications. Since each state in $M(X)$ may be mapped to states in multiple SMEDL monitors, we borrow the idea of *labelled FSM* [114] to label each state with sets of parameters and transitions in it expand the sets of parameters in the target state based on the parameters in the transition. A labelled FSM $LM(X)$ is a five-tuple $\langle LSt, \Sigma(X), (\iota, \{\}), \phi, LF\rangle$ where $LSt = St \times 2^X$ is a set of *labelled states* and LF and ϕ are defined as the smallest sets satisfying the following relation:

$$\langle (q, S), e(\overline{x}), (q', S \cup \overline{x})\rangle \in \phi \ if \ \langle q, e(\overline{x}), q'\rangle \in \sigma$$

$$(q, S) \in LF \ if \ q \in F$$

We use *Example 1* to give an intuition of the algorithm. The labelled FSM is shown in Figure 7 (a). In the remainder of this chapter, all shaded states are final states while white ones are non-final states. All self-looping transitions are omitted in the figure. The correspondence between the labelled FSM and the SMEDL specification (illustrated in Figure 7 (b)) is represented by numbers marked at the transitions. The idea for our algorithm is that we identify connected components in $LM(X)$ with the same set of parameters in labels and factor them out into separate SMEDL monitors, along with transitions between the states in the connected component. Transitions between the connected components are turned into events transferred between the SMEDL monitors, captured by the monitoring architecture. Due to the feature of labelled FSMs, the target state of a transition must have equal or more parameters than the source state. For instance, transitions (4), (5) and (6) and all implicit self-looping transitions connect component with the same set of parameters. Transitions (1), (2) and (3), on the other hand, expand the set of parameters, which are transformed into two transitions as shown Figure 7 (b). The one in the monitor with less parameter information (fewer number of parameters) raises an event to trigger the transition in the monitor with more information.

When feeding the event trace $\tau_0 : updateM(m_1), createC(m_1, c_1), createC(m_2, c_2), createI(c_1, i_1), useI(i_1)$ [101] to the SMEDL monitors, state update of the monitor mc and mci is given in

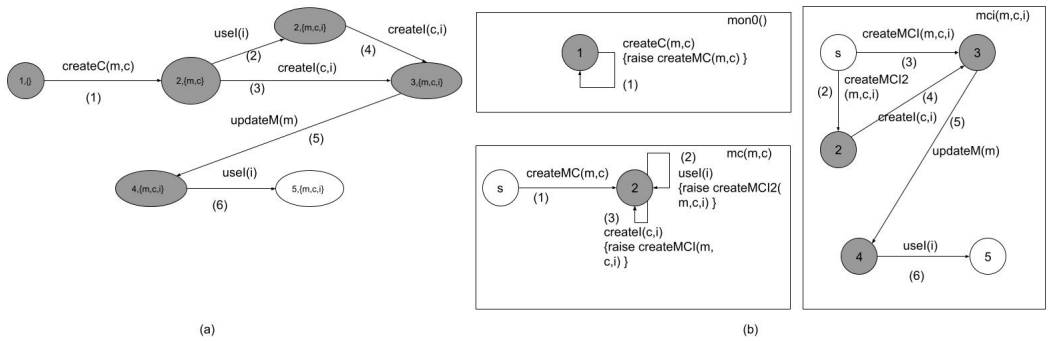

Figure 7: Labelled FSM and SMEDL monitors for Example 1

Table 4. Note that $mon0$ is omitted because it has only one instance and always stays at the initial state. In MOP, each event is equipped with a field *flag* to indicate whether this event can start a monitoring process. As stated in the specification [101], $updateM$ does not start the process, thus no instance is created. $createC(m_1, c_1)$ and $createC(m_2, c_2)$ are received by $mon0$, which creates two instances of mc. $createI(c_1, i_1)$ triggers the creation of $mci(m_1, c_1, i_1)$ by sending *createMCI*(i_1) to mci. $mci(m_1, c_1, i_1)$ is in state 3 after creation. $useI(i_1)$ triggers a self-loop transition in $mci(m_1, c_1, i_1)$ and creates an instance of $mci(m_2, c_2, i_1)$ by extending $mc(m_2, c_2)$.

Table 4: State update of SMEDL monitors given τ_0

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

The formal transformation process is given in Algorithm 2, which creates a SMEDL specification $S(W)$ from $LM(W)$. $S(W)$ is a tuple $\langle SMon, arch\rangle$ where $SMon$ is a map from a set of parameters to the corresponding parametric monitors and $arch$ is an architecture description. A single SMEDL monitor in $S(W)$, $smon(W')$, is an FSM $\langle St_{smon}, \Sigma_{smon}, \iota_{smon}, \psi_{smon} : St_{smon} \times \Sigma_{smon} \rightarrow St_{smon} \times (\Sigma(W) \cup \{null\}), F_{smon}\rangle$ where $W' \subseteq W$. Each transition in ψ_{smon} takes the form $\langle q, e(\overline{x}), q', r\rangle$ where r is either an event to be raised after the transition or null. We use the dot operator to access the elements of tuples. It is worth noting that the SMEDL specification introduced here is compatible with the definition given in Chapter 3: the FSM is a scenario in each monitor and the parameter set W' corresponds to the field $paras$ of the monitor interface in Section 3.2.2, but without order because each parameter has a name and the order does not matter.

The process performs by traversing transitions in $LM(W)$. Note that f is the flag to denote whether the event can start the monitoring process. From line 4 to 10, the SMEDL monitor $mon1$ parameterized by W_1 is obtained by either from $SMon$ or creating a new one. When creating a new monitor type, its initial state is ι of LM if W_1 is empty; otherwise, a dummy initial state $s0$ is created. For the SMEDL with no parameters (denoted as mon_0), only one instance will be created at the beginning of the execution. As a result, its execution starts with ι. Monitors with non-empty parameters are created from mon_0 or other monitors and the creation event will trigger the transition from the initial state $s0$ in the created instance as shown below.

If W_1 and W_2 are identical (line 11 to 17), the transition is directly mapped to $mon1$. If W_1 is not identical to W_2, instances of $mon2(W_2)$ are created by instances of $mon1(W_1)$ through e. The condition at line 19 is used to avoid creating $mon2$ from mon_0 using an event that cannot start the monitoring process. Then, $mon2(W_2)$ is obtained (line 19 to 23). To represent the behavior of extension of parameters, two transitions are added respectively to $mon1$ and $mon2$. The transition added to $mon1$ receives e and raises $crEv$. The event $crEv$ creates an instance of $mon2$ when the instance does not exist and transitions from s_0 to q'. Note that each monitor may have multiple creation events. Each $crEv$ is identified by $\overline{x}$, as stated at line 24. As a side note, the order of monitors in $monTypeList$ can also be determined in the process: $mon2(W_2)$ is put before $mon1(W_1)$ in $monTypeList$. Communication between $mon1$ and $mon2$ through $crEv$ is specified in the architecture description (line 33 - 37). Finally, monitors in $SMon$ are completed (line 38) by adding self-looping dummy transitions.

4.2.2. Correctness proof of transformation

In Algorithm 1, $\Delta : [[X \rightharpoonup V] \rightharpoonup St]$ maps from a parameter instance to its current state. In SMEDL, there is no explicit concept or notation for slicing. Instead, slicing is achieved by dynamic evolution of monitor network. *Configurations* are used represent slices and the corresponding state. We abuse the terminology *configuration* to denote a mapping from monitor instances to St. The range of configuration is St because monitor instances never stay at the initial state except for mon_0, whose initial state belongs to St. The equivalence relation between Δ and *configuration* is

Algorithm 2 Transformation from labelled FSM to SMEDL

```
1: procedure TRANSFORMATION2SMEDL(LM = ⟨LSt, Σ(W), (ι, {}), ϕ, LF⟩)
2:     SMon ← ∅, arch ← ∅
3:     for ⟨(q, W1), e(x̄, f), (q′, W2)⟩ ∈ ϕ do
4:         mon1 ← SMon.get(W1)
5:         if mon1 == null then
6:             if W1 == ∅ then
7:                 mon1 ← ⟨{ι}, ∅, ι, ∅, ∅⟩
8:             else
9:                 mon1 ← ⟨{s0}, ∅, s0, ∅, ∅⟩
10:            SMon.put(W1, mon1)
11:        if W1 == W2 then
12:            mon1.St ← mon1.St ∪ {q, q′}, mon1.Σ ← mon1.Σ ∪ {e(x̄)},
13:            mon1.ψ ← mon1.ψ ∪ {⟨q, e(x̄), q′, null⟩}
14:            if q ∈ LF then
15:                mon1.F ← mon1.F ∪ {q}
16:            if q′ ∈ LF then
17:                mon1.F ← mon1.F ∪ {q′}
18:        else
19:            if W1 ≠ ∅ || f == true then
20:                mon2 ← SMon.get(W2)
21:                if mon2 == null then
22:                    mon2 ← ⟨{s0}, ∅, s0, ∅, ∅⟩
23:                    SMon.put(W2, mon2)
24:                crEv ← (creationEvName(x̄), W2, true)
25:                mon1.St ← mon1.St ∪ {q}, mon1.Σ ← mon1.Σ ∪ {e(x̄)},
26:                mon1.ψ ← mon1.ψ ∪ {⟨q, e(x̄), q, crEv⟩}
27:                mon2.St ← mon2.S ∪ {q′}, mon2.Σ ← mon2.Σ ∪ {crEv},
28:                mon2.ψ ← mon2.ψ ∪ {⟨mon2.ι, crEv, q′, null⟩}
29:                if q ∈ LF then
30:                    mon1.F ← mon1.F ∪ {q}
31:                if q′ ∈ LF then
32:                    mon2.F ← mon2.F ∪ {q′}
33:                patterns ← ∅
34:                for i ∈ range(0, |W1| − 1) do
35:                    patterns ← patterns ∪ {i, mon1, i}
36:                    patterns ← patterns ∪ {i, crEv, i}
37:                arch ← arch ∪ {⟨mon1, crEv, mon2, crEv, patterns⟩}
38:    SMon ← completeFSM(SMon)
39:    return (SMon, arch)
```

defined below. Note that we use $m(x)$ to denote the monitor type m when x is its parameters. An instance of m with binding θ is denoted as $m(\theta)$. If the monitor type is not important, we also use θ to denote the monitor instance.

Definition 31 *Let Δ be a mapping from parameter instances to monitor states of a MOP monitor and config be the configuration of a SMEDL monitor network. Δ is equivalent to config (denoted as $\Delta \equiv$ config) if : 1) for each $\theta \in dom(\Delta)$ and $\Delta(\theta) = s$, there exists an instance $\theta \in dom(config)$ that $config(\theta) = s$; and 2) for each $\theta \in dom(config)$ where $\theta \neq \emptyset$ and $config(\theta) = s$, $\theta \in dom(\Delta)$ and $\Delta(\theta) = s$.*

Suppose a SMEDL specification Mon_{SMEDL} is constructed from a MOP monitor Mon_{mop}, the relation of states between them are described as the theorem below:

Theorem 32 *Given an event trace τ, if Δ is the state of Mon_{MOP} computed by $\mathbb{C}\langle X\rangle$ and config is the state of Mon_{SMEDL} computed according to the semantics of the SMEDL monitor network, then $\Delta \equiv$ config.*

The theorem states that the SMEDL monitor network obtained by the transformation above implements the trace slicing of MOP. The proof of Theorem 32 is performed by induction on the length of trace τ and then comparing the state update between Mon_{mop} and Mon_{SMEDL} given a new incoming event.

Proof (sketch): **Basic step.** Suppose $\tau = \{e(\theta, f)\}$ where θ is the parameter binding and f is the *flag* field mentioned above. If f is true, $\mathbb{C}\langle X\rangle$ creates a mapping $[\theta \mapsto \sigma(\iota, e)]$ and puts it into Δ. If θ is empty, no new instance needs to be created because mon_0, the monitor with no parameters, exists at the beginning of execution. If θ is not empty, mon_0 receives e and creates a monitor instance $mon(\theta)$. In both case, mon_0 and mon will be updated in consistent with σ according to the transformation. If f is false, e is ignored in $\mathbb{C}\langle X\rangle$. In SMEDL, there is no transition triggered by e and no instance is created. Consequently, the equivalence relation holds in the basic step.

Inductive step. Suppose after consuming a trace τ, the state of Mon_{mop} is Δ and the state of

Mon_{SMEDL} is *config* and $\Delta \equiv config$ according to the inductive hypothesis. After an event $e(\theta, f)$ is consumed, Mon_{mop} is updated to Δ' and Mon_{SMEDL} is updated to $config'$. We want to prove that $\Delta' \equiv config'$. There are two cases:

case 1: $\Delta(\theta)$ is defined. According to the inductive hypothesis, for any $\theta_1 \in dom(\Delta)$ and $\Delta(\theta_1) = s_1$, $\theta_1 \in dom(config)$ and $config(\theta_1) = s_1$. If θ_1 is equal to or has more parameter information than θ, we assume $\Delta'(\theta_1) = s_2$ where $s_2 = \sigma(s_1, e)$, denoted as *tr*. According to Algorithm 2 (line 13), tr is directly added to the monitor instance θ_1, which leads to the same state update such that $config'(\theta_1) = s_2$. If θ_1 does not have more information than θ, $\Delta'(\theta_1) = \Delta(\theta_1) = s_1$. A corresponding creation event e' is raised for a monitor $m(dom(\theta_1 \sqcup \theta))$. However, according to Lemma 29, $\Delta(\theta_1 \cup \theta)$ must have been defined so there *must have* already existed a corresponding instance of *m* defined in $config$. e will trigger an equivalent transition in it as stated above while e' will trigger a self-loop transition because a monitor is constructed completely in Algorithm 2. As a result, $\Delta' \equiv config'$.

case 2: $\Delta(\theta)$ is not defined. $\mathbb{C}\langle X \rangle$ first tries to find an existing parameter instance θ' which is maximal among all instances with less parameter information than θ and temporarily set state of θ to $\Delta(\theta')$. If no such θ' is found and f is true, instance of θ is created with the initial state ι. Otherwise, no instance will be created. Then, θ is used to create new instances by extending existing compatible bindings in Δ in reversed topological order of parameter information. Finally, each θ' defined in Δ that has equal or more parameter information than θ are updated with e.

In SMEDL, updating and extending is conducted in the similar way. We need to prove that 1) an equivalent instance for θ is created; and 2) equivalent instances by extending parameter of existing instances are created; 3) updates of instances by $e(\theta, f)$ are also equivalent.

Suppose we have $mon(dom(\theta))$ in *monTypeList*. For 1), we need to consider the case in which f is true, which means there exists a creation event e' raised from mon_0 when it receives e. However, because e is dispatched to monitors in reversed topological order of parameter information enforced in *monTypeList* and mon_0 is the last element in *monTypeList*, $mon(\theta)$ will be created by mon_0 only

when there are no existing instances from which $mon(\theta)$ can be created. Furthermore, the behavior of creating $mon(\theta)$ from mon_0 and updating its state is equivalent to the behavior of Algorithm 1 at line 19 and line 17.

For 2), the target is to prove $dom(\mathit{config}') = dom(\Delta')$. In Algorithm 1, θ is combined with all compatible instances in $dom(\Delta)$. As a result, given an arbitrary instance $\theta' \in dom(\Delta)$, if $\theta \sim \theta' \wedge \theta \not\sqsubseteq \theta'$, a parameter instance $\theta \sqcup \theta'$ is defined by extending from an instance $\theta'' \in dom(\Delta)$ that $\theta \sqcup \theta' = \theta \sqcup \theta''$. Note that θ' may be identical to θ'' or not. Since SMEDL monitors are complete, an instance parameterized by $\theta \sqcup \theta'$ will be generated by extending an existing instance because θ' and θ'' exist in the domain of config due to the inductive hypothesis.

For 3), we need to prove the state equivalence between them. For an arbitrary instance $\theta' \in dom(\Delta')$, there are two cases. If $\theta' \in dom(\Delta)$, $e(\theta)$ update its state when $\theta \sqsubseteq \theta'$. Line 12 and 13 in Algorithm 2 guarantees that $\mathit{config}'(\theta') = \Delta'(\theta')$. If θ' is a newly created instance, it must take the form $\theta_1 \sqcup \theta$ where θ_1 is either empty or $\theta_1 \in dom(\Delta)$. We have $\Delta'(\theta') = \sigma(\Delta(\theta_2), e)$ where $max(\theta_2, dom(\Delta), \theta')$ because the traversal is performed in the reverse topological order of the parameter information. For SMEDL, multiple creation events may be raised to create an instance parameterized by θ'. The inductive hypothesis and the semantics of the monitor network guarantees that it is extended from θ_2. Since $\mathit{config}(\theta_2) = \Delta(\theta_2)$, $\mathit{config}'(\theta') = \sigma(\Delta(\theta_2), e) = \Delta'(\theta')$. $\square$

Discussion on the specification size. Each MOP specification is transformed into a set of SMEDL monitors. Suppose there are n parameters, there are at most 2^n SMEDL monitor specifications. Although the size of specification is exponential to the number of parameters, it is still acceptable because the number of parameters for a parametric property is usually small.

4.3. Expressing trace slicing of QEA using SMEDL

A QEA $Q(\Lambda)$ contains two parts. Q is an EFSM and $\Lambda \in (\{\forall, \exists\} \times vars(Q))^*$ is a list of quantified parameters (quantifier list for short) where $vars(Q)$ is the set of parameter variables appearing in Q. QEA adopts a similar slicing strategy to MOP. However, unlike MOP, in which parametric property is a mapping from sub traces to verdicts, QEA aggregates the results of all full bindings. The

interpretation of a quantifier list in QEA leads to a significant difference between QEA and MOP in generating bindings: QEA stores any binding that can be built from the derived domain that has a non-empty projection.

Example 2 (CandidateSelection) [14]: 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, denoted as Q_{can}, is illustrated Figure 8 (a). 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. Self-looping transitions are omitted. 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, which hints that *candidate* is the event to start the monitoring process.

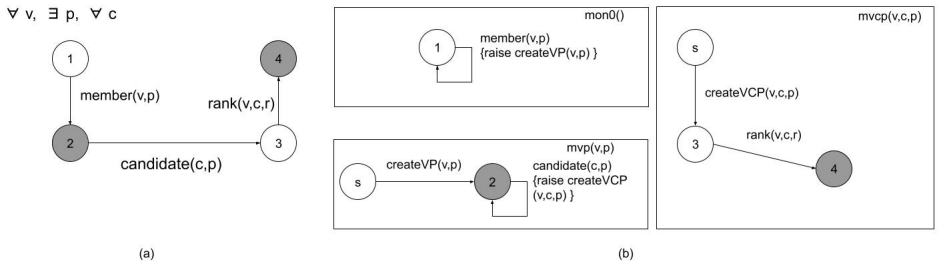

Figure 8: QEA and SMEDL specification of CandidateSelection

When feeding the event trace τ: *member*(*tom*,*red*), *member*(*ali*,*blue*), *candidate*(*jim*,*red*), *candidate*(*flo*,*red*), *candidate*(*don*,*blue*), *rank*(*tom*,*jim*,1), *rank*(*ali*,*don*,1), *rank*(*tom*, *flo*, 2), Q_{can} generates the set of full bindings as shown in Table 5. QEA generates bindings from combinations across the domain of each parameter, except for ones that are not compatible with any incoming events. Consequently, 9 full bindings are generated by Q_{can}. Then the overall verdict is computed by aggregating the verdicts of each full binding based on the big-step semantics. However, not all bindings represent the genuine relation between voters, parties and candidates. For instance, the triple (tom, red, don) states that don belongs to the party red, which is not true for τ.

We transform the FSM of Q_{can} into the SMEDL specification S_{can}, as shown in Figure 8 (b). Since *candidate* is the only event to start the monitoring process, there is no specification for (c, p) or (v, c). When feeding τ, S_{can} only generates three bindings, as illustrated in Table 6. We can observe

voter	party	candidate	state
tom	red	jim	4
tom	red	flo	4
tom	red	don	2
tom	blue	don	1
ali	red	jim	1
ali	red	flo	1
ali	blue	jim	2
ali	blue	flo	2
ali	blue	don	4

Table 5: Bindings generated by QEA [14]

voter	party	candidate	state
tom	red	jim	4
tom	red	flo	4
ali	blue	don	4

Table 6: Bindings generated by SMEDL

that these three bindings correctly represent the relations between voters, parties and candidates. Furthermore, if we perform proper logical aggregation over these three bindings, as stated below, the identical verdict with the QEA monitor can be obtained. This result hints that SMEDL monitors may be able to monitor properties of QEA while generating fewer bindings.

In this section, we present a transformation from QEA (denoted as $Q(\Lambda)$) to SMEDL (denoted as $S(\Lambda)$). Each SMEDL monitor in S is denoted as $qMon(l)$ where $l \subseteq \Lambda$. The transformation contains two steps, transformation of the EFSM and the quantified parameter list. For the EFSM, Algorithm 2 is directly used. The semantics of quantifiers over parameters is encoded as a SMEDL monitors for aggregation. The intuition is to generate aggregation monitors for each parameter, which are connected in a list according to the position in the quantifier list. Verdicts will be grouped according to the parameter values of bindings that generate them. The universal and existential quantifier are respectively implemented by conjunction and disjunction over grouped verdicts.

We encode the algorithm of MOP for trace slicing in SMEDL. As a result, an equivalent relation between QEA and SMEDL specifications cannot be built in general. However, we specify two subsets of QEA specifications that can generate identical verdicts. In the first case (Section 4.3.1), if all parameters are universally quantified and all bindings that are not created in the SMEDL monitor are guaranteed to stay at a final state (the verdict returns true), the SMEDL monitor only needs to aggregate the verdicts of full bindings it maintains. In the second case (Section 4.3.2), if all bindings that are not created in the SMEDL monitor are guaranteed to stay at a non-final state, we can get identical results through enhancing aggregation monitors for all universally quantified variables by checking the size of the domain and the number of values appearing. If two values are

different, it means there would exist a binding that is only maintained by the QEA monitors. Since the corresponding verdict is false, the final verdict is also false.

4.3.1. Encoding aggregation semantics in SMEDL

The big-step semantics of QEA can be described as a function $Verdict_{QEA}(Q, \tau) \equiv \Pi_{v_1 \in dom(p_1)} \Pi_{v_2 \in dom(p_2)} ... \Pi_{v_n \in dom(p_n)}(F(\Delta_Q^\tau(v_1, v_2, ..., v_n)))$ where $(v_1, v_2, ..., v_n) \in dom(\Delta_Q^\tau)$. Π_{p_x} is $\wedge$ ($\vee$) when the corresponding quantifier of a parameter variable p_x in Q is $\forall$ ($\exists$). By default, $dom(\Delta_Q^\tau) \equiv dom(p_1) \times ... \times dom(p_n)$. We abuse the notation Δ_Q^t as the mapping from bindings to states when fed with the trace t. t may be omitted when the context is clear. We also reuse the symbol F and define $F(s)$ to map a state s to true (false) if it is a final (non-final) state.

Similarly, $Verdict_{SMEDL}$ computes the verdict from the SMEDL specification S: $Verdict_{SMEDL}(S, \tau) \equiv \Pi_{v_1 \in dom(p_1)} \Pi_{v_2 \in dom(p_2)} ... \Pi_{v_n \in dom(p_n)}(F(config_S^\tau(v_1, v_2, ..., v_n)))$ where $(v_1, v_2, ..., v_n)$ is a full binding and $(v_1, v_2, ..., v_n) \in dom(config_S^\tau)$. $config_S^\tau$ is the mapping from bindings to the states when fed with the trace τ. The domain of $config_S^\tau$ contains generated full bindings. When the context is clear, τ is omitted.

The abstract function $Verdict_{SMEDL}$ is implemented as a set of aggregation monitors. Intuitively, we group full bindings that has the same value over the sub list of parameters $p_1, p_2, ..., p_{n-1}$. Logical conjunction (disjunction) is performed over the verdicts of bindings in the same group when p_n is universally (existentially) quantified. The verdicts obtained at this level will be further grouped based on the value of sub list $p_1, p_2, ...p_{n-2}$. This process is repeated in a hierarchical way until getting the final verdict. Following this idea, each aggregation monitor takes the form $aMon_{p_x}(p_1, p_2, ...p_{x-1})$ for each parameter p_x except for p_1 because $aMon_{p_1}()$ has no parameters.

The $aMon_{p_x}(p_1, ..., p_{x-1})$ is an FSM $\langle St_{smon} : \{\iota_1, \iota_2\}, \Sigma : \{count_x, result_x\}, \iota_{smon} : \iota_1, \sigma_{smon}, F_{smon} : \{\iota_1\}\rangle$. Two input events $count_x$ and $result_x$ are sent from the upstream aggregation monitor $aMon_{p_{x+1}}$. $count_x$ indicates the number of results to receive while each $result_x(..., b)$ carries the verdict value b of a binding where the values of $p_1, p_2, ..., p_{x-1}$ match the parameter values of $aMon_{p_x}$. For the universal quantifier, the conjunction over verdicts carried in the $result_x$ is defined

Table 7: Transitions of $aMon_{p_x}(p_1, ..., p_{x-1})$ for the universal quantifier

source	event	guard	action	target
ι_1	$count_x(p_1, .., p_x)$	$count_{p_x} == 0$	$count_{p_x} = 1;\ raise\ count_{x-1}(p_1, ..., p_{x-1});$	ι_1
ι_1	$count_x(p_1, .., p_x)$	$count_{p_x}! = 0$	$count_{p_x} ++;$	ι_1
ι_1	$result_x(p_1, ..., p_x, b)$	$count_{p_x} > 1$ && b	$count_{p_x} --;$	ι_1
ι_1	$result_x(p_1, ..., p_x, b)$	$count_{p_x} == 1$ && b	$count_{p_x} --;\ raise\ result_{x-1}(p_1, ..., p_{x-1}, b)$	ι_1
ι_1	$result_x(p_1, ..., p_x, b)$	!b	$count_{p_x} --;\ raise\ result_{x-1}(p_1, ..., p_{x-1}, b)$	ι_2

as the transition set σ_{smon} in Table 7. The action in the first transition indicates that $count_{x-1}$ is the creation event of $aMon_{p_{x-1}}$. The aggregation for the existential quantifier can be defined in a similar way with disjunction over verdicts collected from the upstream monitor.

The aggregation monitors are chained in the reversal order of Λ from the fully-bound monitor $qMon(\Lambda)$ down to $aMon_{p_1}()$. During execution, whenever a new instance of $qMon(\Lambda)$ is created, corresponding aggregation monitor instances are created or updated. The specification of $qMon(\Lambda)$ is updated as follows: 1) for each state s, a transition $(s, end(), s, \{result_n(p_1, ..., p_n, b)\})$ is added where b is true/false when s is a final/non-final state; and 2) for each transition triggered by a creation event, an action of raising the event $count_n(p_1, ..., p_n)$ is added. Upon receiving the event end, verdicts generated by instances of $qMon(\Lambda)$ are sent to corresponding instances of $aMon_{p_n}$ to triggering the computation of aggregation.

Figure 9 illustrates the architecture of the monitor network of S_{can}. $mon0$, mvp and $mvcp$ correspond to the monitors in Figure 8. The $aMon_c(v, p)$ and $aMon_v()$ monitors perform the conjunctions to implement the universal quantifier for c and v while $aMon_p(v)$ implements the semantics of the existential quantifier for p. After merging the verdicts generated by the bindings illustrated in Table 6 using these aggregation monitors, we can get the same verdict with QEA.

Comparison of SMEDL and QEA. The difference between $Verdict_{QEA}$ and $Verdict_{SMEDL}$ comes from creation and maintenance of bindings. Given a trace τ, $bd_{SMEDL}(S(\Lambda), \tau)$ and $fullbd_{SMEDL}(S(\Lambda), \tau)$ respectively denote the set of monitor instances and instances bound with all parameters generated from S by consuming τ. $bd_{QEA}(Q(\Lambda), \tau)$ and $fullbd_{QEA}(Q(\Lambda), \tau)$ denote the set of bindings and full bindings in QEA. When the context is clear, Λ and τ are omitted. It is obvious that $bd_{SMEDL}(S) \subseteq bd_{QEA}(Q)$ and $fullbd_{SMEDL}(S) \subseteq fullbd_{QEA}(Q)$. $infer(Q, S, \tau)$ ($fullInfer(Q, S,$

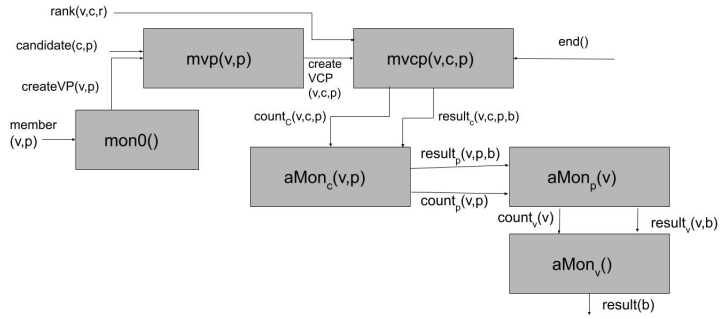

Figure 9: Connections between SMEDL monitors for Example 2

τ)) are *inferred* (full) bindings that do not have corresponding monitor instances in SMEDL.

The relation between QEA and SMEDL are studied by comparing $Verdict_{QEA}$ and $Verdict_{SMEDL}$. Lemma 33 states that for a binding that appears in both SMEDL and QEA, the state of that binding between SMEDL and QEA are identical.

Lemma 33 *Suppose $Q(\Lambda)$ is a QEA specification and $S(\Lambda)$ is the corresponding SMEDL specification. For each binding $\theta \in bd_{SMEDL}(S, \tau)$, $config_S(\theta) == \Delta_Q(\theta)$.*

Proof (sketch): by Theorem 32, we already proved that $dom(config_S) = dom(\Delta)$ and for each binding $\theta \in dom(\Delta)$, $\Delta(\theta) = config_S(\theta)$ where Δ is the mapping from bindings to states in Algorithm 1. Since QEA follows the same semantics on updating existing bindings and creating new bindings from existing ones, we have $\Delta_Q(\theta) = \Delta(\theta)$ for each $\theta \in dom(\Delta)$. Moreover $bd_{SMEDL}(S, \tau) \subseteq dom(\Delta_Q)$. As a result, for each binding $\theta \in bd_{SMEDL}(S, \tau)$, $config_S(\theta) = \Delta_Q(\theta)$. □

Using Lemma 33, we only need to consider the state of inferred bindings. Whether $Verdict_{QEA}$ and $Verdict_{SMEDL}$ generate the identical result depends on the characteristics of the inferred bindings. Lemma 34 states that for a QEA specification Q of which all parameters are universally quantified, if all inferred full bindings are created and stay at a final state by consuming an input trace τ, Q and S generates the same verdict for τ.

Lemma 34 *For a QEA specification $Q(\Lambda)$ of which all parameters are universally quantified and*

its corresponding SMEDL monitor S, given an input trace τ, if $\forall\theta \in fullInfer(Q, S, \tau)$, $\Delta_Q(\theta) == true$, then $Verdict_{QEA}(Q, \tau) == Verdict_{SMEDL}(S, \tau)$.

Proof (sketch): because all quantifiers are universal, and the verdict of all inferred bindings are all true, both $Verdict_{SMEDL}$ and $Verdict_{QEA}$ are equivalently reduced to computing conjunction over bindings belonging to $fullbd_{SMEDL}(S(\Lambda), \tau)$, which means $Verdict_{QEA}(Q, \tau) = Verdict_{SMEDL}(S, \tau)$. $\square$.

To prove Theorem 37, we first prove Lemma 35 and Lemma 36, which respectively state the relation between inferred bindings and SMEDL monitor instances and the property of the inferred full bindings.

Lemma 35 *Given a QEA specification $Q(\Lambda)$, its corresponding SMEDL specification S and an input trace τ, for a binding $\theta \in infer(Q, S, \tau)$ that does not project to an empty trace, if $\Delta_Q^\tau(\theta) == s$, there must exist a binding W that $max(W, bd_{SMEDL}(S, \tau), \theta) \wedge config_S^\tau(W) == s$.*

Proof (sketch): The proof is performed by induction on the input trace τ.

Base step: if τ has only one event $e(\theta')$, there are no inferred bindings.

Inductive step: given an input trace τ_0, $\exists\theta \in infer(Q, S, \tau_0)$ that $\Delta_Q^{\tau_0}(\theta) = s$. There exists a binding W that $max(W, bd_{SMEDL}(S, \tau_0), \theta) \wedge config_S^{\tau_0}(W) = s$. When an event $e(\theta')$ arrives, we first consider cases where θ' is treated as a whole. τ is obtained by appending e to τ_0.

1) $\theta' \sqsubseteq \theta$: e triggers a transition for θ that $\Delta_Q^\tau(\theta) = s'$. For W, if $\theta' \sqsubseteq W$, the same transition is triggered so that $config_S^\tau(W) = s'$ and $max(W, bd_{SMEDL}(S, \tau), \theta)$ still holds because for any other binding $W' \in bd_{SMEDL}(S, \tau_0) \wedge W' \sqsubseteq W$, $W' \sqcup \theta' \sqsubseteq W$. Otherwise, a new binding $W' \equiv W \sqcup \theta' \sqsubseteq \theta$ is created and $config_S^\tau(W') = s'$. Since W is maximal in S with respect to θ and $\theta' \sqsubseteq \theta$, $max(W', bd_{SMEDL}(S, \tau), \theta)$ still holds.

2) $\theta \sqsubseteq \theta'$: e will not update θ in Q and W in S and no other bindings that have less information in W will be updated either.

3) $\theta' \sim \theta$: a new binding $\theta \sqcup \theta'$ is created in Q and $\Delta_Q^\tau(\theta \sqcup \theta') = s'$. In S, A new binding $(W \sqcup \theta') \sqsubseteq (\theta \sqcup \theta')$ is created and $\mathit{config}_S^\tau(W \sqcup \theta') = s'$. Moreover, $max(W \sqcup \theta', bd_{SMEDL}(S, \tau), \theta \sqcup \theta')$ holds.

4) $\theta' \not\sim \theta$: θ' will not trigger the transition for θ so both $\Delta_Q^\tau(\theta)$ and $\mathit{config}_S^\tau(W)$ are equal to s. $max(W, bd_{SMEDL}(S, \tau), \theta)$ still holds because $W' \sqcup \theta' \not\sqsubseteq \theta$ for any binding $W' \sqsubseteq \theta$ that is compatible with θ'.

QEA also creates bindings by adding a subset of θ'. Suppose a binding $\theta'' \sqsubseteq \theta'$ is compatible with θ while θ' is not. A new inferred binding $\theta \sqcup \theta''$ is created and $\Delta_Q(\theta \sqcup \theta'') = s$ because e is not projected to this binding. There is no corresponding monitor instance $\theta \sqcup \theta''$ created in S. As a result, $max(W, bd_{SMEDL}(S, \tau), \theta \sqcup \theta'')$ holds and $\mathit{config}_S^\tau(W) = s$. $\square$.

Lemma 36 *Given a QEA specification $Q(\Lambda)$ of which all parameters are universally quantified and its corresponding SMEDL specification S, if*

$$\forall \lambda \in dom(S.SMon) \wedge \lambda \subset \Lambda, S.SMon(\lambda).St_{smon} \backslash \{s0\} \subseteq S.SMon(\lambda).F_{smon},$$

then $\forall \theta \in \mathit{fullInfer}(Q, S, \tau), F(\Delta_Q^\tau(\theta)) == true$ for any arbitrary input trace τ.

<u>Proof (sketch)</u>: the proof is performed by induction on the length of τ.

Base step: when the event trace has only one event $e(\theta')$, there is no inferred full bindings.

Inductive step: suppose $\forall \theta \in \mathit{fullInfer}(Q, S, \tau_0), F(\Delta_Q^{\tau_0}(\theta)) = true$ holds. When an event $e(\theta')$ arrives, we analyze the evolution of $\mathit{infer}(Q, S)$ and $\mathit{fullInfer}(Q, S)$. We use τ to denote τ_0 appended with e. As stated above, $e(\theta')$ may 1) update state of an existing binding; 2) create a new binding of θ'; 3) create new bindings by extending existing bindings.

For 1), if e updates a partial binding or the target state is a final state, nothing changes; if e updates a full binding $\theta \in \mathit{fullInfer}(Q, S, \tau_0)$ by a transition tr from a final state s to a non-final state s' such that $\Delta_Q^\tau(\theta) = s'$, we know that based on Lemma 35, there exists a binding W in S that

$max(W, bd_{SMEDL}(S, \tau_0), \theta) \wedge config_S^{\tau_0}(W) = s$. Then, given $e(\theta')$, a new instance $W' \equiv W \sqcup \theta'$ is created in S and $config_S^{\tau}(W') = s'$. Since a non-final state only appears in the fully-instantiated monitor, θ' and W are compatible with θ, $W' = \theta$, which means $\theta \notin fullInfer(Q, S, \tau)$. As a result, the post condition holds for τ.

For 2), according to the slicing semantics of QEA and SMEDL, we need to find two bindings W_S and W_Q that satisfy the predicate $max(W_S, bd_{SMEDL}(S, \tau_0), \theta')$ and $max(W_Q, bd_{QEA}(S, \tau_0), \theta')$. We only need to consider the case in which $W_Q \in infer(Q, S, \tau_0)$. Based on Lemma 35, we know that $W_S \sqsubseteq W_Q$ and $\Delta_Q^{\tau_0}(W_Q) = config_S^{\tau_0}(W_S)$. When fed with $e(\theta')$, new bindings $W_Q \sqcup \theta' = W_S \sqcup \theta' = \theta'$ are created. Neither the domain or range of $fullInfer(Q, S, \tau)$ will not be updated. As a result, the post condition holds for τ.

For 3), $e(\theta')$ can extend bindings in $infer(Q, S, \tau_0)$ in two ways. We will use θ'' to denote the generated binding. In the first way, θ'' is created by treating θ' as a unbreakable entity following the slicing strategy used in SMEDL. The projected sub trace of θ'' will contain e. If θ'' is a non-full binding or $F(\Delta_Q^{\tau}(\theta'')) = true$, nothing changes. If $F(\Delta_Q^{\tau}(\theta'')) = false$, we can use the similar way presented in 1) to prove that $\theta'' \in bd_{SMEDL}$ so that $\theta'' \notin fullInfer(Q, S, \tau)$. In the second way, θ'' is created from an existing binding $\theta \in bd_{QEA}$ that is not compatible with θ'. Therefore, θ'' is an inferred binding and the project sub trace for θ'' will not include e. $\Delta_Q^{\tau}(\theta'') = \Delta_Q^{\tau_0}(\theta)$ holds. In both ways, the post condition holds. □

Theorem 37 *Given a QEA specification $Q(\Lambda)$ of which all parameters are universally quantified and its corresponding SMEDL specification S, if*

$$\forall \lambda \in dom(S.SMon) \wedge \lambda \subset \Lambda, S.SMon(\lambda).St_{smon} \setminus \{s0\} \subseteq S.SMon(\lambda).F_{smon}$$

then $Verdict_{QEA}(Q, \tau) \equiv Verdict_{SMEDL}(S, \tau)$ for any arbitrary input trace τ.

Proof (sketch): since Q only contains universally quantified parameters, $Verdict_{QEA}$ and $Verdict_{SMEDL}$ are computed by conjunction over verdicts of all generated full bindings. Lemma 33 indicates that all shared full bindings between Q and S generate identical verdicts. From Lemma 36,

we know that verdicts of all inferred full bindings are true. As a result, $Verdict_{QEA}(Q, \tau) = Verdict_{SMEDL}(S, \tau)$ for any arbitrary input trace τ because of Lemma 34. □

Note on Example 2. Example 2 does not satisfy the syntactic restrictions on the precondition of Theorem 37. We give a trace that leads to different results in QEA and SMEDL: $member(v1, p1)$, $candidate(c1, p1)$, $member(v1, p2)$. QEA outputs true while SMEDL outputs false on this trace. The reason is that a referred full binding $(v, p2, c1)$ is created in QEA and stays at the final state 2. To achieve equivalent result in SMEDL, we need to add a transition in $mvp(v, p)$ to trigger the creation of instances of $aMon_c(v, p)$ so that the verdict of a non-full binding $mvp(v1, p2)$ can be counted. As a future work, we will modify the definition of $Verdict_{SMEDL}$ and the aggregation monitor to relax syntactic restriction on QEA so that more QEA properties such as Example 2 can be monitored using SMEDL.

Discussion on the specification size and memory overhead of SMEDL monitors. Same with the transformation of MOP, the upper bound of the number of SMEDL monitors transformed from the event automaton is exponential to the number of parameters. The number of aggregation monitors is equal to the number of parameters. During execution, the memory overhead of SMEDL monitors include partial and full bindings and instances of aggregation monitors. As stated above, the number of full bindings of QEA is asymptotically $O(n_{p_1} * n_{p_2} * ... * n_{p_k})$ where n_{p_k} is the domain size of parameter p_k. For SMEDL, the upper bound of number of full bindings is the same but in most cases fewer bindings would be generated, especially when parameters are related. For instance, each voter or candidate only corresponds to one party in *Example 2*. The number of full bindings is about $O(n_{voter} * (n_{candidate}/n_{party}))$ given that the number of candidates for each party is the same. The number of instances for the aggregation monitors is proportional to the number of full bindings and each instance maintains state variables such as $count$ and $result$. Consequently, when parameters are related by the semantics defined in events, SMEDL may maintain fewer instances, potentially leading to less memory overhead.

sender	receiver	state
A	B	4
A	C	4
B	C	4

Table 8: Bindings generated using SMEDL

sender	receiver	state
A	B	4
A	C	4
B	B	2
B	C	4

Table 9: Bindings generated using QEA

4.3.2. Enhancement of aggregation monitors

In this section, we consider cases where all inferred bindings are guaranteed to stay at a non-final state at the end of execution. We modify the aggregation monitor to count the domain of each parameter to infer the state of bindings that are not available for the SMEDL specification during runtime.

Example 3 (Broadcast) [114]: for every sender s and receiver r, after s sends a message it should wait for an acknowledgement from r before sending again. The QEA and SMEDL specification are shown in Figure 10. Suppose the input trace τ' is $send(A)$, $send(B)$, $ack(B, A)$, $ack(C, A)$, $ack(C, B)$, SMEDL and QEA respectively generate three and four full bindings, as shown in Table 8 and Table 9. Because the binding $[r \mapsto B, s \mapsto B]$ stays at a non-final state, the verdict generated by SMEDL and QEA are different. The least information needed for the SMEDL specification to generate the identical verdict includes 1) the size of the derived domain for the receiver and 2) the implicit knowledge on the verdict of all inferred bindings. For 1), we define $counter$ monitors to keep track of the derived domain for parameters. For 2), we impose syntactic restrictions on the QEA specification such that all inferred bindings always stay at a non-final state.

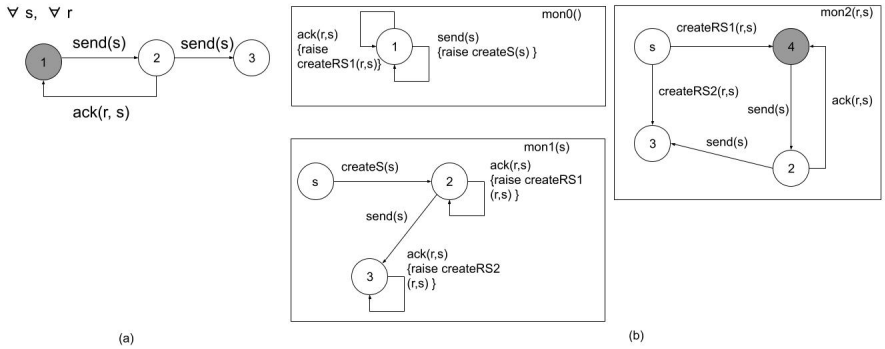

Figure 10: QEA and SMEDL specification for Broadcast

Table 10: Transitions of $frontend(p_x)$

source	event	guard	action	target
ι_1	$e(.., p_x, ...)$	true	$raise\ add_{p_x}();$	ι_1

Table 11: Transitions of $backendp_x()$

source	event	guard	action	target
ι_1	$add_{p_x}()$	true	$dom\_size_{p_x} ++;$	ι_1
ι_1	$end()$	true	$raise\ size_{p_x}(dom\_size_{p_x});$	ι_1

Counter monitors. For each universally quantified parameter p_x, we define a counter monitor $cMon_{p_x}$ to keep track of the domain size of p_x. The $cMon_{p_x}$ monitor is divided into two monitors. As stated in Table 10, the monitor $frontend(p_x)$ receives all input events of S containing p_x as parameters and creates new instances whenever a new value of p_x is observed. The $backend_{p_x}$ monitor keeps track of the domain size of p_x by receiving add_{p_x} raised by each instance of $frontend(p_x)$. The domain size is sent to the corresponding aggregation monitor $aMon_{p_x}$ by receiving the event end.

The $aMon_{p_x}$ monitor also needs to be modified as shown in Table 12. Two added transitions are triggered by $size_{p_x}(dom\_size_{p_x})$ which carries the size of the domain of p_x raised from $backendp_x()$. If the size of the domain is not equal to the number of values bound in the bindings generated by S, the result will be false. $size_{p_x}$ must be received before any $result$ events so that comparison of domain size can be done before aggregating the result. To enforce this order, the event *end* is delivered to the counter monitor, which sends $size_{p_x}$ to the corresponding aggregation monitor $aMon_{p_x}$. Then, one of the aggregation monitor raises the $trigger$ event to trigger the execution of $qMon(\Lambda)$.

Figure 11 illustrates the communication pattern among functional and aggregation monitors. The diagram $broadcast(r, s)$ represents the functional monitors illustrated in Figure 10 (b). Note that $mon2(r, s)$ is modified to receive the $trigger$ event to output events $count_r$ and $result_r$ to $aMon_r$.

To analyze the behavior of the enhanced aggregation monitor, we first update the definition of $Verdict_{SMEDL}$ by adding the operation on checking the size of derived domain against the number of

Table 12: Added transitions of $aMon_{p_x}(p_1, ..., p_{x-1})$

source	event	guard	action	target
ι_1	$size_{p_x}(dom\_size_{p_x});$	$count_{p_x} == dom\_size_{p_x}$	$raise\ trigger()$	ι_1
ι_1	$size_{p_x}(dom\_size_{p_x});$	$count_{p_x} != dom\_size_{p_x}$	$raise\ result_{x-1}(p_1, ..., p_{x-1}, false)$	ι_3

values observed in the generated bindings: $Verdict^{dom}_{SMEDL}(S,\tau) \equiv \Pi_{v_1 \in dom(p_1)} ... \Pi_{v_n \in dom(p_n)}(F(configs_S(v_1, v_2, ..., v_n))) \wedge_{p_x \in uni_Q} (\wedge_{inst \in aMon_{p_x}.inst}(inst.count_{p_x} == inst.dom\_size_{p_x}))$ where $(v_1, v_2, ..., v_n) \in fullbd_{SMEDL}(S(\Lambda), \tau)$. uni_Q denotes the set of universally quantified parameters of Q. The symbol $aMon_{p_x}.inst$ denotes the set of instances of $aMon_{p_x}$. The symbol $count_{p_x}$ and $dom\_size_{p_x}$ are the state variables defined in $aMon_{p_x}$ above. The aggregation monitors with counter monitors implement $Verdict^{dom}_{SMEDL}$.

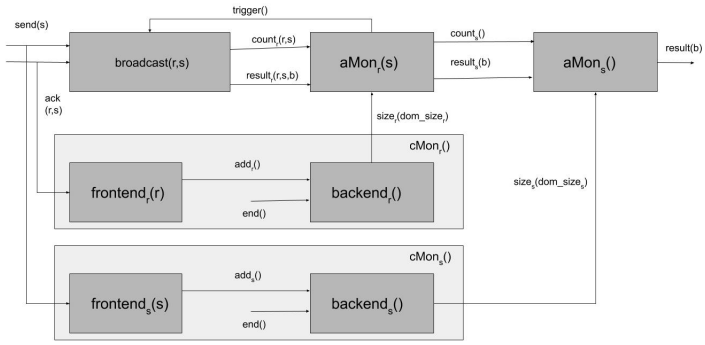

Figure 11: Communication architecture of monitors for broadcast

Similar to the previous section, we can prove Theorem 38 that $Verdict^{dom}_{SMEDL}(S,\tau)$ and $Verdict_{QEA}(Q,\tau)$ can generate the identical result on an arbitrary input trace τ when the QEA specification Q is transformed into S where all partial monitors only have are non-final states. Note that because Example 3 satisfies the syntactic restriction on the QEA specification in this theorem, the QEA and the SMEDL monitors can always generate identical results.

Theorem 38 *Given a QEA specification $Q(\Lambda)$ and its corresponding SMEDL specification S, if*

$$\forall \lambda \in dom(S.SMon) \wedge \lambda \subset \Lambda, \forall st \in S.SMon(\lambda).St_{smon}, st \notin S.SMon(\lambda).F_{smon}$$

then $Verdict_{QEA}(Q,\tau) == Verdict^{dom}_{SMEDL}(S,\tau)$ for any arbitrary input trace τ.

To prove Theorem 38, we prove the relation between $Verdict^{dom}_{SMEDL}$ and inferred full bindings in Lemma 39. Then in Lemma 40, we prove that $Verdict_{QEA}(Q,\tau) = Verdict^{dom}_{SMEDL}(S,\tau)$ if all inferred full bindings return false.

Lemma 39 *For a QEA specification $Q(\Lambda)$ with at least one universally quantified parameter and the corresponding SMEDL monitor S, given an input trace τ, $fullbd_{QEA}(Q,\tau) \neq \emptyset$ iff there exists a universally quantified parameter p_x of which an instance $inst$ of its aggregation monitor satisfies the property that $inst.count_{p_x} \neq inst.dom\_size_{p_x}$.*

Proof (sketch): ($\Rightarrow$): suppose $inst.count_{p_x} \neq inst.dom\_size_{p_x}$ for an instance $inst \equiv aMon_{p_x}(v_1, v_2, ..., v_{x-1})$ where v_1 to v_{x-1} are values for parameter p_1 to p_{x-1} that appear in front of p_x in the parameter list. There must exist a value v_x in the domain of p_x that there are no full bindings of which the parameter values are v_1 to v_x for p_1 to p_x. According to the big step semantics of QEA, these bindings will be generated and they belong to $fullbd_{QEA}(Q,\tau)$.

($\Leftarrow$): Suppose $\theta \equiv (v_1, ..., v_{n-1}, v_n) \in fullbd_{QEA}(Q,\tau)$ where n is the length of Λ. There is no monitor instance in $qMon(v_1, ..., v_n) \in S$. Since $count_{p_n}$ for the instance $aMon_{p_n}(v_1, ..., v_{n-1})$ is updated whenever a new instance of $qMon(\Lambda)$ is generated, it will be smaller than $dom\_size_{p_n}$.
□

Lemma 40 *For a QEA specification $Q(\Lambda)$ and its corresponding SMEDL specification S, given an input trace τ, if $\forall\theta \in fullInfer(Q,S,\tau), \Delta_Q(\theta) == false$, then $Verdict_{QEA}(Q,\tau) == Verdict^{dom}_{SMEDL}(S,\tau)$.*

Proof (sketch): if $fullInfer(Q,S,\tau) = \emptyset$, $Verdict^{dom}_{SMEDL}(S,\tau) \equiv \Pi_{v_1 \in dom(p_1)} \Pi_{v_2 \in dom(p_2)} \cdots \Pi_{v_n \in dom(p_n)}(F(config_S(v_1, v_2, ..., v_n)))$. Since $Verdict^{dom}_{SMEDL}$ and $Verdict_{QEA}$ are computed based on the same set of full bindings, they are equivalent.

If $fullInfer(Q,S,\tau) \neq \emptyset$ and there is at least one universally quantified variable, according to Lemma 39, $Verdict^{dom}_{SMEDL}$ and $Verdict_{QEA}$ are both evaluated to false. In the case where all variables are existentially quantified and all inferred full bindings return false, $Verdict^{dom}_{SMEDL}(S,\tau)$ is true if there is a binding in θ such that $config_S(\theta) = true$. Based on Lemma 33, $\Delta_Q(\theta) = true$ so $Verdict_{QEA}(Q,\tau) = true$. If there is no binding θ such that $config_S(\theta) = true$, corresponding bindings in Q are false. Since all inferred full bindings are also false, $Verdict_{QEA}(Q,\tau) = false$.
□

Similar to Lemma 36, Lemma 41 is proved by induction on the input trace. Since final states can only appear in the fully-bound monitor in S, if a full binding is created or transitions to a final state in Q, there must exist a corresponding monitor instance in S. Theorem 38 can then be proved by using Lemma 40 and Lemma 41.

Lemma 41 *Given a QEA specification $Q(\Lambda)$ and its corresponding SMEDL specification S, if*

$$\forall \lambda \in dom(S.SMon) \wedge \lambda \subset \Lambda, \forall st \in S.SMon(\lambda).St_{smon}, st \notin S.SMon(\lambda).F_{smon}$$

then $\forall \theta \in fullInfer(Q, S, \tau)$, $F(\Delta_Q^\tau(\theta)) == false$ for any arbitrary input trace τ.

Proof (sketch): the proof can be performed in a similar way with Lemma 36 by induction on τ. □

Discussion on the specification size and memory overhead of SMEDL monitors. We need to consider the counter monitors $cMon$. For the specification size, in the worst case, each parameter corresponds to one $cMon$. During execution, number of instances for the counter monitors is asymptotically $O(n_{p_1} + n_{p_2} + ... + n_{p_k})$, which is smaller than multiplication over the sizes of each parameter when most parameters have a domain of which the size is greater than 1.

4.4. Summary

In this chapter, we presented a novel method to encode trace slicing using a SMEDL monitor network. We first proved that SMEDL monitor network can express the efficient slicing algorithm of MOP by proposing a transformation from MOP and SMEDL. Then, we defined two syntactic fragments of QEA from which equivalent SMEDL monitors can be generated. As a future work, it would be useful to formalize the relationship with MOP and QEA in Coq. By transforming into SMEDL monitors, QEA properties can be efficiently monitored. In Chapter 5, we will perform experiments to illustrate time efficiency of using SMEDL to monitor parametric properties. Due to the limitation of the language design, SMEDL cannot express all QEA properties. By analyzing Example 2, however, we see potential to further loosen the syntactic restrictions to support more QEA properties. We also 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.

CHAPTER 5 : Implementation and Evaluation

In Chapter 3, we formally defined the language and generate correct-by-construction Haskell implementation for single monitors using Coq. We have also developed a compiler that generates executable C code from SMEDL specifications [5]. Full language features such as monitor network and flexible deployment are supported. As for the time efficiency, the C monitor is 10 time faster than the Haskell version for the case study in Section 3.1.5. In this chapter, we will first present the implementation of a synchronous set (Section 5.1). Then, we compare our technique to representative tools (Section 5.2) with respect to time efficiency. By profiling execution of monitors, we discover bottleneck of the monitor code and propose intuitions for optimization from both the perspective of language design and implementation (Section 5.3). We will also study how monitor synchronous and asynchronous deployment influence the time overhead (Section 5.4). SMEDL specifications of benchmarks used in this Chapter are given in the Appendix A.2.

5.1. Implementation of synchronous sets

In SMEDL, multiple monitors can be organized as a synchronous set. For implementation, communication among monitors in the same synchronous set is implemented as function calls which use queue to control event dispatching. Figure 12 illustrates the architecture of a synchronous set. Execution of a synchronous set is controlled by a global wrapper, which realizes the semantics proposed in Section 3.2. A global wrapper maintains two queues, *InnerQueue* and *OutputQueue*. *InnerQueue* stores events that are consumed within the synchronous set while *OutputQueue* stores events that will be sent to the environment. Exported events raised during execution of monitors are added to the *InnerQueue* or/and the *OutputQueue* accordingly. When an event *e* from the environment is enqueued to the *InnerQueue*, the global wrapper fetches *e* from the head of the *InnerQueue* and dispatches it to monitors that can handle it. As stated in Chapter 4, if multiple monitors can handle *e*, calls are made sequentially according to the order specified in *monTypeList*. This calling order guarantees that a new instance of monitor is always created from an existing instance with the most

[5] https://github.com/PRECISE/SMEDL

informative identities, which is necessary for parametric monitoring. Each monitor handles e by its *local wrapper*, which tries to retrieve monitor instances from the *instance store* that can handle e according to the architecture description. If there is no existing instance that handles e and the event connection is unicast, a new instance is created by the *local wrapper* and inserted to the store. Then, the local wrapper updates the state of each fetched instance by executing the monitoring logic. Events raised from instances are enqueued to *InnerQueue* if they are to be consumed by monitors in the synchronous set or the *OutputQueue* if they are sent to monitors in a different synchronous set or just raised as an alarm. The execution continues until the *InnerQueue* becomes empty, meaning that all monitors have finished their execution in reaction to the arrival of e. Then events in the *OutputQueue* will be sent out to the environment by the *global wrapper*.

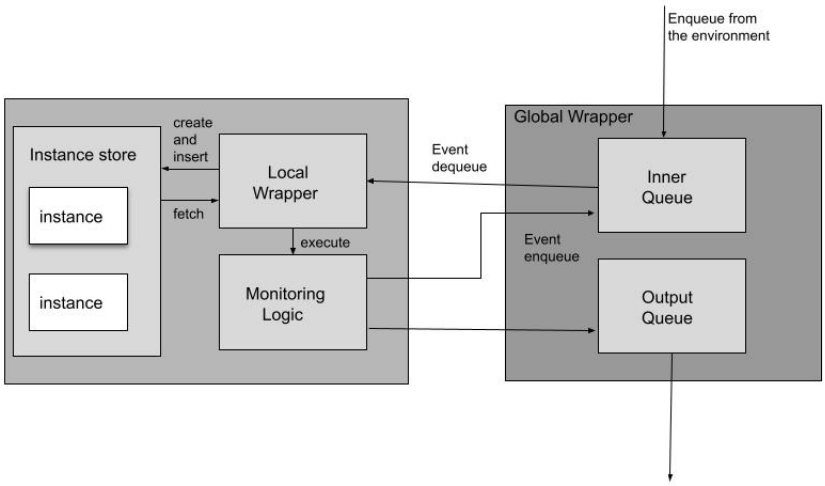

Figure 12: Architecture of synchronous set

Instance store. As shown below, retrieving and inserting of instances are intensive. To achieve efficient monitoring, it is important to make each such operation fast, especially when there are million of instances to maintain. If all instances are stored in a linear data structure, the average retrieving time would be quadratic to the number of instances if the number of instances is also linear to the length of the trace. Instead, we use binary trees to store references to the monitor instance, as illustrated in Figure 13. For each parameter p, an AVL tree is maintained. Each node in the tree is the head node of a list of references to the instances that have the same value on p. For each node, all its descendant nodes on its left sub tree have less identity value while the right

sub tree stores nodes linking to instances with larger identity value. When the local wrapper tries to retrieve an instance (or multiple instances), it starts with searching the AVL tree of a non-wildcard parameter. Then, linear search is performed to filter based rest of the parameters. As illustrated in the experiment below, tree structures lead to much better performance than the linear structure.

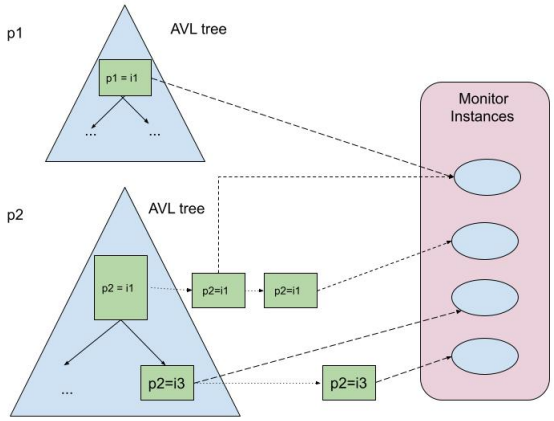

Figure 13: Using AVL trees as instance store

5.2. Tool evaluation

This section evaluates the SMEDL framework by comparing to RV-monitor [99] for online monitoring and QEA and MonPoly [19] for offline monitoring. Implemented in Java, RV-monitor uses the same formalism as JavaMOP. It can generate C and Java monitors for online monitoring. Implemented in OCaml, MonPoly checks properties described in MFOTL formulas. All experiments in this section are done under the following platform: Xeon(R) Gold 6148 CPU at 2.40GHz, 750GB memory, running Ubuntu 18.04 LTS 64-bit operating system.

5.2.1. Evaluation of online monitoring

Online evaluation is performed over three benchmark programs: *Watertank*, *BasicCar* and *UnsafeFile*.

- The program *Watertank* is a controller of water tank, which simulates the behavior of updating water tanks based on the commands read from the environment. The monitor checks whether the incoming commands follow the operational policy such as the valve of a tanker cannot be

opened when the tank is not in service.

- The program *BasicCar* simulates the behavior of a car such as start, stop, acceleration, toggle of lights and wipers. The monitor checks whether the program follows the policy of car operation.

- The program *UnsafeFile* opens a series of files and write texts into them. The monitor checks whether all file descriptors are closed at the end of execution and no writing operations are performed after the file is closed.

For each program, we measure execution time of the original program and monitors at 10 different input scales (number of instances), which is linear to the size of the event trace and the number of instances created. At each input, the result is obtained by averaging over 10 executions. The result is shown in Table 13, Table 14 and Table 15. We can observe that the runtime overheads of SMEDL monitors in these examples are reasonable because 1) SMEDL monitors are efficient and 2) instrumentation sites in these programs are sparse. As for comparison with RV-monitor, SMEDL performs better than RV-monitor in *Watertank* for all inputs except for the case where the instance number is 1000. Moreover, as shown in Figure 14, Figure 15 and Figure 16, SMEDL monitors achieve nearly linear increment of execution time along with the length of the trace while RV-monitors are quadratic because it uses lists to store instances.

Table 13: Execution time of SMEDL and RV-monitor in *Watertank*

trace size	instance number	original program time(s)	SMEDL time(s)	RV-monitor time(s)
1495	1000	9.718	0.011	0.007
3084	2000	18.956	0.023	0.030
4535	3000	30.120	0.036	0.065
5935	4000	40.640	0.050	0.117
7532	5000	53.210	0.062	0.181
8909	6000	66.152	0.076	0.262
10510	7000	80.260	0.092	0.357
11946	8000	95.765	0.104	0.468
13489	9000	110.270	0.119	0.590
14960	10000	126.530	0.132	0.734

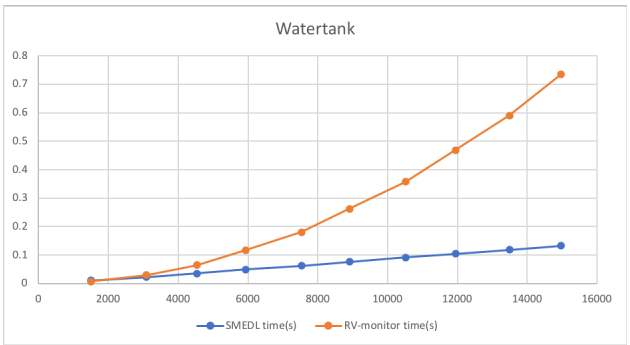

Figure 14: Execution time of SMEDL and RV-monitor in *Watertank*

Table 14: Execution time of SMEDL and RV-monitor in *UnsafeFile*

trace size	instance number	original program time(s)	SMEDL time(s)	RV-monitor time(s)
0.5M	500	0.722	0.278	0.928
1M	1000	1.534	0.566	3.706
1.5M	1500	2.247	0.868	8.336
2M	2000	2.998	1.188	14.709
2.5M	2500	4.154	1.474	22.366
3M	3000	5.043	1.789	30.793
3.5M	3500	5.998	2.199	40.126
4M	4000	6.998	2.591	50.206
4.5M	4500	9.285	2.815	61.371
5M	5000	10.526	3.171	72.628

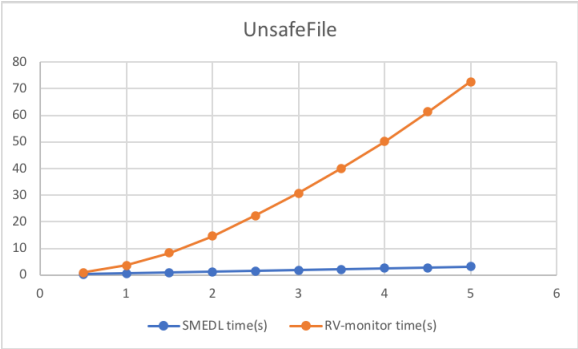

Figure 15: Execution time of SMEDL and RV-monitor in *UnsafeFile*

5.2.2. *Evaluation of offline monitoring*

Offline monitoring is performed over 7 properties against QEA and 3 properties against MonPoly. All properties and traces are from the competition on runtime verification in 2014[6] and 2016.[7]

- *QEA-GrantCancel*: every resource should only be held by at most one task at any one time

[6]https://gitlab.inria.fr/crv14/benchmarks/-/tree/master/OFFLINE
[7]https://crv.liflab.ca/wiki/index.php/Offline_track

Table 15: Execution time of SMEDL and RV-monitor in *BasicCar*

trace size	instance number	original program time(s)	SMEDL time(s)	RV-monitor time(s)
60000	10000	0.722	0.018	0.637
120000	20000	1.366	0.036	4.607
180000	30000	2.003	0.055	12.850
240000	40000	2.708	0.072	23.031
300000	50000	3.251	0.095	38.015
360000	60000	4.055	0.113	54.345
420000	70000	4.614	0.132	76.007
480000	80000	5.233	0.155	99.731
540000	90000	5.859	0.178	130.110
600000	100000	6.562	0.192	155.580

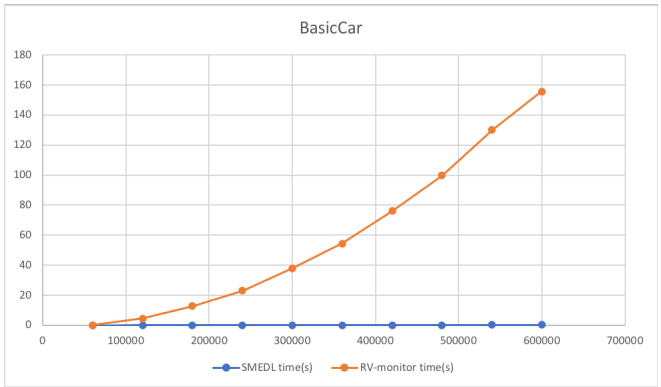

Figure 16: Execution time of SMEDL and RV-monitor in *BasicCar*

and if a resource is granted to a task, it must be cancelled before being granted to another task.

- *QEA-NestedCommand*: every command issued later must succeed before previously issued ones.
- *QEA-ResourceLifeCycle*: a resource goes through three stages of free, requested and granted in sequence.
- *QEA-RespectConflict*: resources that are in conflict with each other cannot be granted at the same time.
- *QEA-Auction*: the auction process follows certain policies such as: 1) each bid is strictly larger than the previous bid; 2) an item is sold if the last bid amount is greater than the initial price; 3) an item can only be sold in the auction for once; and so on.

- *QEA-CandidateSelection*: *Example 2* in Section 4.3.
- *QEA-SqlSanitizer*: every string derived from an input string is sanitized before use.
- *MonPoly-Banking-1*: executed transactions of any customer must be reported within at most 5 days if the transferred money exceeds a given threshold of $2,000.
- *MonPoly-Banking-2*: executed transactions of any customer must be authorized by some employee between 2 to 20 days before they are executed if the transferred money a exceeds a given threshold of $2,000.
- *MonPoly-Publish*: 1) any report must be approved prior to its publication; 2) the person who publishes the report must be an accountant and the person who approves the publication must be the accountant's manager; 3) the approval must happen within at most 10 days before the publication.

We transform QEA specifications and MFOTL formulas into SMEDL monitors. The transformation from QEA has been discussed in Chapter 4. There is no existing work on transformation from MFOTL formulas into EFSMs so we manually translate these examples by considering the intention of the property and the semantics of MFOTL. Correctness of the transformation is justified by testing against several traces. The size of the SMEDL specification for the benchmark properties are illustrated in Table 16. The column *monitor#* represents the number of monitors in the specification; *max parameter #* represents the max number of parameters among all monitors; *connection #* represents the number of connections defined in the architecture file. We can observe that *CandidateSelection*, *SqlSantitizer* and *Publish* are more complicated than others in number of monitors or event connections.

The experiment is performed by measuring execution time and maximum memory allocated for monitors during execution. The comparison between SMEDL and QEA is illustrated in Table 17. Among 7 properties, SMEDL outperforms QEA in 6 of them in time efficiency and SMEDL uses less memory than QEA. Although it is not totally fair to compare monitors implemented in C and

Table 16: Size of SMEDL specifications

property	monitor#	max parameter #	connection #
QEA-GrantCancel	1	1	2
QEA-NestedCommand	2	2	6
QEA-ResourceLifeCycle	1	1	6
QEA-RespectConflict	2	2	6
QEA-Auction	1	1	4
QEA-CandidateSelection	4	3	10
QEA-SqlSanitizer	2	2	11
MonPoly-Banking-1	1	1	2
MonPoly-Banking-2	1	1	2
MonPoly-Publish	6	3	16

Java, the result still demonstrates that SMEDL is competitive tool for offline monitoring.

Table 17: Comparison between SMEDL and QEA

property	trace size	QEA time(s)	QEA mem (KB)	SMEDL time	SMEDL mem (KB)	speedup
GrantCancel	1M	2.357	1079924	1.994	1874	1.18
NestedCommand	1200	1.371	537018	0.018	1897	76.17
ResourceLifeCycle	1M	2.487	1364046	1.915	2749	1.30
RespectConflict	1M	3.887	1429354	3.123	1966	1.24
Auction	84643	0.436	149437	0.251	2440	1.74
CandidateSelection	977997	26850	37973789	3135	2127146	8.56
SqlSanitizer	9447751	9.6	1929383	41.413	1843685	0.23

The comparison between SMEDL and MonPoly is illustrated in Table 18. The result shows that SMEDL performs better or approximately equally in *Banking-1* and *Banking-2* where the size of SMEDL specification is small. For *Publish*, MonPoly performs much better than SMEDL. In the next section, we explore some directions to further optimize SMEDL monitors.

Table 18: Comparison between SMEDL and MonPoly

property	trace size	MonPoly time(s)	MonPoly mem(KB)	SMEDL time(s)	SMEDL mem(KB)	speedup
Banking-1	320424	4.270	20281	0.951	11052	4.49
Banking-2	323308	0.703	14319	0.785	11843	0.90
Publish	57404	14.375	416793	1517	1003576	0.01

5.3. Optimization

Figure 12 indicates that execution of a monitor network can be divided into three disjoint parts: operations on the global queue; store operations, including fetching and insertion of created instances; and execution of monitoring logic. To analyze the bottleneck of monitor performance, we profiled execution of SMEDL monitors to analyze how each part contributes to the execution time. Two representative patterns are discovered, as shown in the results of two benchmarks *UnsafeMapIter*

(*Example 1* in Chapter 4) and *UnsafeFile* respectively in Figure 17 and Figure 18. The X-axis represents the overall execution time of monitors while the Y-axis represents the number of fully-bound instances created during execution. Note that the trace length is linear to the number of instances. We can observe that in *UnsafeMapIter*, insert and fetch operation take respectively 50% and 25% of the overall execution time; in *UnsafeFile*, fetch operation takes 55% of the overall execution time while insertion operation can be negligible. The reason is that in *UnsafeMapIter*, both partial and fully-bound instances are created and each instance is queried only once. In *UnsafeFile*, on the other hand, each monitor instance is queried one thousand times because of the writing operation. However, in both cases, store operations dominate more execution time than queue and monitoring logic, which admits some optimization possibility.

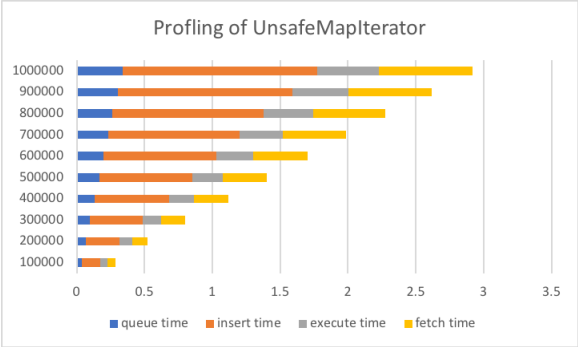

Figure 17: Profiling of the SMEDL monitor for UnsafeMapIter

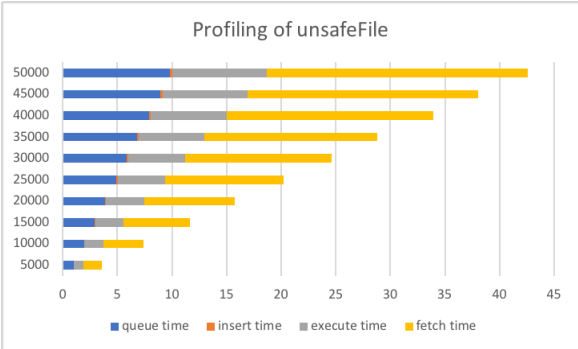

Figure 18: Profiling of the SMEDL monitor for UnsafeFile

5.3.1. Optimization intuitions

The overall execution time of store operations depend on two factors: 1) the number of operations to execute and 2) the time to execute each operation. For 1), we introduce *explicit* creation events that avoid useless fetch operations; for 2), we introduce the concept of *final* states to remove useless instances. At the implementation level, we use the hash map to achieve (nearly) constant time insertion and retrieval of instances.

Avoidance of unnecessary retrieving operations. To motivate this feature, we recall the *CandidateSelection* property, which requires that for each voter, there must exist a party of which the voter is a member and all candidates of that party must be ranked by the voter. The example trace we used in the experiment [8] is distributed as follows: the first 343,135 *member(v,p)* indicate the relation between the voter (v) and the party (p); then the next 6,759 *candidate(c,p)* represent the information between the candidate (c) and the party (p). Because there are no duplicated events, the triple *(v,c,p)* is created by the only pair of *member* and *voter*. After all triples are created, the rest are *rank(v, c, r)* for ranking the candidate c as r by the voter v. The SMEDL specification contains one partial monitor $mvp(v, p)$ and one full monitor $mvcp(v, p)$. Instances of $mvcp$ is created by receiving the event $createVCP$. In the implementation, $createVCP$ is an *implicit* creation event, which means it will trigger the local wrapper of $mvcp$ to retrieve a matching instance before creating it. However, if the trace does not contain duplicated events, $createVCP$ always creates a new instance because no such instance has been created. As a result, the verdict would not change if we remove retrieving operation in the local wrapper when receiving $createVCP$. We incorporate this change to the code and the execution time is 19 minutes, which is much faster than the original implementation. To generalize this idea, we extend the architecture description language with *explicit* creation events. In contrast to *implicit* creation events, which creates an instance after failing to retrieve an instance, an explicit creation event directly creates an instance without triggering execution of monitoring logic. Formal definition and use of an explicit creation event and analysis on how the time overhead would be improved are left as future work.

[8] https://crv.liflab.ca/wiki/index.php/Offline_Team2_Benchmark2

Final states. When the execution time of retrieving an instance depends on the size of the instance store, it is desirable to maintain as few instances as possible. We introduce the concept of *final* states, which are specified for a scenario. For an instance, if all scenarios in which final states are defined reach to a final state,it will be automatically removed. To evaluate the effect of final states, we experiment against *NestedCommand* and *GrantCancel*. The experimental result is illustrated in Table 19. For *NestedCommand*, the monitor with final states performs better because fewer instances are maintained during execution. In contrast, the monitor with final states performs worse for *GrantCancel*. The reason is that the event to grant a resource appears repeatedly in the trace. As a result, monitor instances that have been removed may be created later, which leads to more store operations. From these two examples, we could find that if a monitor can reach to a sink state and the trace does not have any event that may trigger creation of this instance after it is removed, setting sink states as final states can improve time efficiency.

Table 19: Comparison of monitors with/without final states for NestedCommand and GrantCancel

Property	trace length	without final states(s)	with final states(s)	speedup
NestedCommand	500	0.008	0.005	1.60
NestedCommand	1200	0.018	0.007	2.57
GrantCancel	100000	0.25	0.28	0.89
GrantCancel	500000	1.02	1.13	0.90

Hash map for instance store. We have used AVL trees as the data structure to store instances, which performs much better than linear search. However, when retrieving an instance by multiple parameters, only the first parameter is searched through the tree while others are filtered in a linear way, which would lead to significant performance degradation. To overcome this drawback, we implement hash maps to store monitor instances, as shown in Figure 19. Suppose the monitor has two parameter variables, p and q, For each subset of non-wildcard parameter combinations, p, q and (p, q), a hash map is created. Within a hash map, all the instances with the same identities in the subset are stored in a list (represented by dot line arrows). For the hashing mechanism, *robin hood* hashing [33] with linear probe is adopted. To compared two ways of instance store, suppose an event with p and q arrives. The monitor using the tree would search from either the tree of p or q to obtain a list of instances. Then, linear search is performed on the list using the value of the other parameter. In contrast, the monitor with the hash map would directly query the hash map for

the pair of p and q. From this example, we can observe that the hash map implementation is more efficient for searching among multiple parameters because it avoids unnecessary linear search and the search space is smaller.

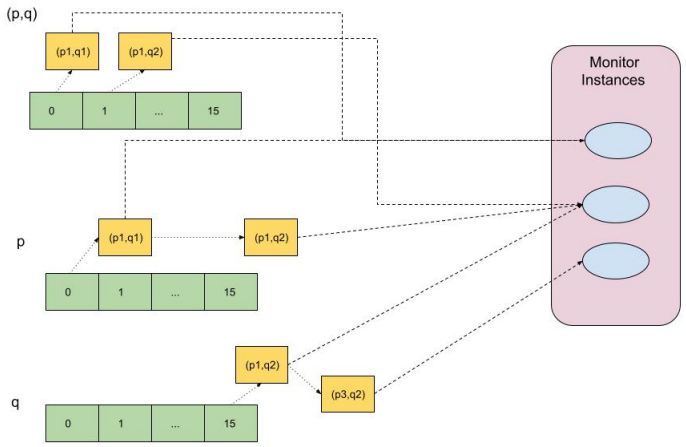

Figure 19: Using hash map instance store

To demonstrate this observation, we perform experiments over 6 case studies on the execution time, as shown in Table 20. The hash map version outperforms the original version in all of them. For *CandidateSelection* and *Publish*, the performance improvement is significant while the boost for *Banking-1* and *Banking-2* are small. The extent of improvement is related to the operations on retrieving instances by multiple parameters. In *Banking-1* and *Banking-2*, instances are retrieved by one parameter. In contrast, there are respectively 4 and 3 monitors in the specification of *Publish* and *CandidateSelection* that have multiple parameters, which lead to many more operations on linear search. In general, we can expect more performance gain when the specification is more complicated with respect to the number of monitors with multiple parameters.

Table 20: Comparison between the tree and hash map implementation on time efficiency

property	trace size	tree time(s)	hash map time(s)	speedup
Auction	84643	0.251	0.23	1.09
CandidateSelection	977997	1141.0 (explicit creation events)	16.02	71.22
SqlSanitizer	9447751	41.413	25.56	1.62
Banking-1	320424	0.951	0.827	1.15
Banking-2	323308	0.785	0.78	1.01
Publish	57404	1517	1.974	768.49

5.4. Evaluation of monitor deployment on time overhead

For online monitoring, monitors may be deployed with the target program synchronously or asynchronously. For synchronous deployment, the target system needs to pause the execution and transfer control to the monitor thus any property violation can be detected timely. For asynchronous deployment, on the other hand, the target system proceeds its execution after invoking APIs to deliver events to the monitor using asynchronous communication. Asynchronous monitoring is useful when the monitor needs to receive events from sources that do not work in the same process or even the same machine. Moreover, asynchronous monitoring may also provide with possibility to reduce the time overhead [32]. The SMEDL framework provides flexible ways to deploy monitors. Synchronous and asynchronous communication are respectively implemented as API calls and RabbitMQ communication middleware. This section compares performance of synchronous and asynchronous monitoring with respect to time efficiency. All experiments in this section are done under the following platform: 2.5 GHz Intel Core i7, 16GB memory, running Ubuntu 18.04 LTS 64-bit operating system.

The evaluation is performed agains three benchmark programs, *UnsafeGrant*, *UnsafeMapIter* and *TrackQuality*. The property *UnsafeGrant* states that a source can only be released after it has been granted and all resources are released at the end of execution. The program *TrackQuality* simulates behavior of generating track data from multiple sources, which has been introduced in Section 3.2.4. The monitor computes statistics for each track in a sliding window divided by timestamps of events and merge them together to check the quality of sensor data.

The experiment measures execution time of synchronously instrumented monitors and calling the RabbitMQ API respectively. The result shown in Table 21 reveals that synchronous monitoring incurs less time overhead in all benchmarks. Calling the RabbitMQ API and related preparation actions such as generation of a message are more time-consuming than monitor execution on average when handling each event. However, the ratio of asynchronous time to synchronous time varies among them. For *UnsafeGrant*, synchronous monitoring is 20 times faster than asynchronous communication while the difference in *TrackQuality* is narrower. The specification of *UnsafeGrant* only

has one monitor specification in which all transitions do not contain any actions such as updates of state variables or arithmetic computations. Moreover, almost all events delivered to the monitor have only one receiving instance. The specification of *TrackQuality*, on the other hand, has 6 monitors and one event can trigger multiple transitions in different monitors with arithmetic and state update actions. As result, synchronous monitoring is less efficient for *TrackQuality*. In general, multiple factors such as the monitor structure, the distribution of events and the time overhead of calling monitoring or communication APIs may influence the time overhead of online monitoring.

Table 21: Comparison of synchronous and asynchronous monitoring

	trace length	sync time(s)	async time(s)	async/sync
UnsafeGrant	60000	0.060	1.206	20.10
UnsafeGrant	150000	0.143	2.777	19.42
UnsafeMapIter	60000	0.052	0.756	14.54
UnsafeMapIter	150000	0.140	1.919	13.71
TrackQuality	1200300	2.553	30.765	12.05
TrackQuality	2000500	4.187	49.860	11.91

5.5. Summary

In this chapter we presented the implementation of the SMEDL framework and demonstrated time efficiency of our technique on large-scaled input by comparing with representative RV tools for both online and offline monitoring. We then profiled execution of monitors and proposed intuition of optimization from the perspective of language and data structure design. Finally, we compare the current implementation of synchronous and asynchronous monitoring, which could give us hints on how to decide monitor deployment with awareness of time overhead.

CHAPTER 6 : Monitoring Time Interval

Timing properties describe the behavior of one event occurring after another event within certain time bound or counting the number of events that occur during an interval of time. In both cases, a monitor needs to not only evaluate the logic of the property but also determine whether events fall within a given time interval. Monitoring timing properties is challenging in the situation where the target system and the monitor are deployed in an asynchronous environment. The asynchronous approach makes monitoring more difficult, due to the network delay and the difference between the system and the monitor clocks. However, by using the monitor clock that is different from the system clock, we may be able to detect that timing behavior of the target system is incorrect because the system clock is wrong.

Although SMEDL monitoring systems supports asynchronous deployment of monitors, explicit clocks cannot be explicitly expressed. In this chapter, we propose a method to clearly separate monitoring of time intervals from the rest of property checking. With this framework, SMEDL can focus on describing monitoring logic without worrying about clocks.

Compatible with SMEDL, we assume the property is checked in an event-driven fashion. To enable checking of the timing in this way, we extend the set of events with a new kind of event that represents the end of a time interval, which we call *interval closure*. Now, we can reduce time checking to *temporal ordering*: if a system event arrives before the closure event, it occurred within the time interval, while if the closure event arrives first, the system event is outside of the interval. In order to produce closure events in the right order, we introduce the *interval handler* module into the monitor.

The second aspect addressed is the design of the interval handler. We note two particular design considerations for the handler: one is *correctness* and the other is *timeliness*. On the one hand, the handler needs to correctly monitor intervals, in the sense that it should close an interval – that is, raise the closure event – only after any event occurring within the interval has been received. In the presence of uncertainty, correct monitoring is possible only if the handler waits long enough to make sure it has seen all relevant events. On the other hand, closing the interval too late may

increase unnecessary resource consumption for monitoring, which should be avoided. Moreover, we should know what the tight one is, in order to be certain that the deadline to be set is larger than the tight one. It is therefore important to set the monitoring *deadline* as small as possible under the premise that correctness of the closure is guaranteed.

To summarize, this chapter addresses the following problem: *"Given an asynchronous environment with uncertain communication delay and imperfect clock synchronization between target system and monitor, under what conditions can correctness of monitoring time intervals be ensured and how to achieve it?"*

We consider three parameters of monitoring setup, network delay, clock skew and clock rate, and study how they influence monitoring time intervals. We explore the parameter space and present a scheme for setting the deadline of monitoring for each interval. We then introduce an algorithm that the interval handler uses to monitor intervals.

Related work. Sammapun [120] considers properties represented with time-bound operators and analyzes several different implementations of checking properties based on timer and heartbeats with bounded or unbounded network delay. However, clock rate and clock skew were not taken into consideration. Lee and Davidson [90] propose algorithms for implementing timed synchronous communication among processes having different clocks such that all processes will decide whether the communication is successful within their own absolute deadlines and they agree on the same decision. Two communication schemes, multiple senders with one receiver and N-way communication were analyzed. They further analyze the performance of two algorithms of timed synchronous communication using probabilistic models [91]. Pinisetty et al. [110] propose a paradigm of runtime enforcement using time retardants on events to ensure that a system satisfies timed properties. Jahanian et al. study the runtime monitoring of time constraints specified by RTL (real-time logic) in the distributed real-time system [80]. However, the monitoring procedure of time intervals was not discussed. They further raise the problem of imprecise timestamps of traces influencing the correct verification of the properties specified by MTL (metric temporal logic) formulas [17]. The paper gave the conclusion that certain MTL fragments can be verified by existing monitors for precise

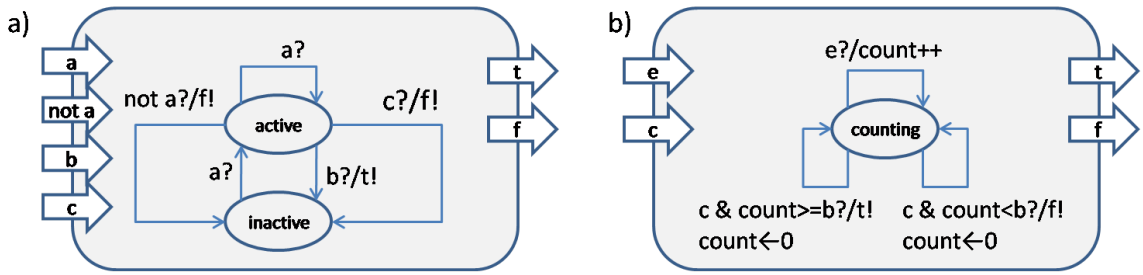

Figure 20: Evaluation of interval operators

traces over traces with imprecise timestamps.

6.1. Motivating examples

Several kinds of commonly used timed specifications involve reasoning over time intervals. We note that, while the logic of evaluating these properties over a stream of events is different, it invariably involves reasoning about intervals of time given in the specification and whether the timestamp of a given observation falls within an interval or outside of it. As we discuss below, parameters of the monitoring setup, such as clock skew or the latency of delivering observations to the monitor, have an impact on how this reasoning should be performed. We therefore want to separate the logic of property evaluation, which depends only on the semantics of the specification language, and interval management, which depends on properties of the monitoring setup.

To illustrate our approach, we first briefly revisit two of them: LTL with interval operators and interval statistics.

LTL with time-bound operators. In LTL, operator Until (U), Weak-until (W) and R (Release) are used to specify properties in a trace. For instance, property $\phi_1 U \phi_2$ is satisfied in a trace if ϕ_1 is satisfied at each location of the trace until ϕ_2 is satisfied at a certain point. The verdict cannot be given to this property until getting the result from the verification of ϕ_2. To restrict the time of getting the result, the time-bound operator is utilized [120]. If we want to express the property that ϕ_2 becomes satisfied within 5 time units from the current time and ϕ_1 remains true within the interval, the formula is written as $\phi_1 U_{[0,5]} \phi_2$.

In many runtime verification approaches [85, 13, 118], temporal operators are evaluated in an event-driven fashion. Arriving events, which could be observations from the target system or results of sub formula evaluation, trigger changes in the operator evaluation status. We want to extend the same approach to interval operators. Consider, for example, evaluation of the bounded-until $aU_{[0,t_1]}b$, where a and b are target system observations. As Figure 20 (a) shows, evaluation of the operator is a state machine that takes as inputs events a, b, and c. Event $not\ a$ represents the absence of a. We refer to the event c as the *interval closure*, which denotes that t_1 time units have elapsed. Note that t_1 is measured in the sense of perfect clock, which may be different from the clock on the system and the monitor side due to the clock skew. Evaluation is activated by an arrival of a, and while further occurrences of a arrive, the state of the evaluation is unresolved. As soon as $not\ a$ arrives, or if the interval is closed, the operator evaluates to false, denoted by raising an event f. But if b arrives before the interval is closed, the operator evaluates to true and an event t is raised. In this way, evaluation of the operator does not depend on the value of the time bound and does not need direct access to the clock. It is straightforward to extend this scheme to cover intervals of the form $[t_1, t_2]$, as well as cover other commonly used temporal operators. Note that to monitor $aU_{[t_1,t_2]}b$ we consider intervals $[0, t_1)$ and $[t_1, t_2]$. When b arrives, we determine, which of the two intervals it falls into, or if it is outside of both. For technical reasons that will be discussed later, we open both intervals when a arrives.

Interval statistics. Some properties needs to collect statistics over a time interval. These properties can be represented in a similar way as SQL queries using aggregate operators [16]. For instance, $Sum_{[0,t_1]}(occur(e)) \ >= \ b)$ specifies the property of the number of occurrences of event e over the time interval [0,t_1] is equal or greater than b. Figure 20 (b) shows the evaluation scheme for this operator in a fashion similar to the previous case. Variable *count* increases with arrivals of event e. When *interval closure* event c arrives, the interval is closed. An event t is raised if *count* is greater or equal than b; otherwise an event f is raised.

In contrast to interval operators discussed above, calculation of interval statistics is different in the sense that intervals are *recurrent*. On the system side, once an interval ends, the next one is

immediately started and statistics calculation continues for the next interval, effectively partitioning the time line into intervals of the same size, starting from some initial event. We can view recurrent intervals as an extension of the two-interval case above.

Checking example. Figure 21 shows a concrete scenario for monitoring of $aU_{[t_1,t_2]}b$ when system events can be delivered with a delay. Assume first that the clocks in both the system and the monitor are perfect. On the monitor side, we begin processing when the event a arrives at relative time 0. To correctly evaluate this property, the monitor needs to tell whether b falls within $i_1 = [0, t_1)$ or within $i_2 = [t_1, t_2]$. Suppose an event b is raised before t_1 but is delayed more than a was and thus arrives after the time t_1 on the monitor side. Thus, at t_1 the monitor cannot yet conclude that i_1 has expired. From the monitor perspective, i_1 and i_2 *overlap*; that is, an incoming event may belong to either interval. However, once we see the timestamp of b, we can tell whether it belongs to i_1 or i_2. Therefore, we do not need to measure duration of i_1 or i_2 on the monitor side. Now consider the case when b does not arrive within i_2. In order to conclude that b did not arrive in time, the monitor has to wait. Eventually, another event with a large enough timestamp may arrive and the monitor may be able to make the conclusion based on that. But what if it arrives after a very long time or, worse, if the missing b was meant to be the last observation? To proceed in a more timely fashion, the monitor has to use a timer. This timer, essentially, sets the *deadline* for b to arrive. This observation underlies our monitoring approach: we use the timer only to safely close the interval, while all other conclusions – whether the interval has started and whether an event is within the interval – are made based on event timestamps.

Apart from the network delay, the clock rate of the system and the monitor also influence interval monitoring. Using the same example above, assume first that there is no clock skew and delivery delay is ranged from 0 to 1 in the sense of the perfect clock. Suppose the clock rate of the perfect clock r_p is 1, clock rate of the system r_s is 0.5 and clock rate of the monitor r_m is within range $[0.8, 1.5]$. Interval i_1 to be monitored is $[0, 6]$ measured by the perfect clock and the monitor begins monitoring it at time 0. To guarantee that all events occurring in i_1 arrive before the monitor finishes monitoring this interval, the deadline of monitoring is set at time 10.5 of the monitor clock, as in the

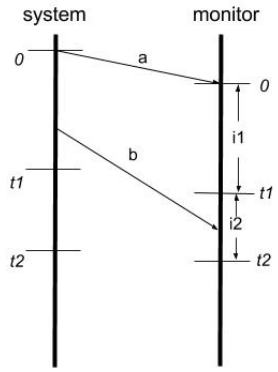

Figure 21: Monitoring time intervals of $aU_{[t_1,t_2]}b$

worst case, an event occurs at 3 of system clock (corresponding to time 6 in the sense of the perfect clock as the clock rate is 0.5) arrives at time 10.5 of the monitor clock with the largest delay. If the actual rate of r_m is 1.5, when an event b happens at time 3.1 of the system clock and the network delay is 0.2 then, it arrives at the monitor at time 9.6 (calculated by 3.1*3 + 0.2*1.5). However, since we know the clock rate of the system is 0.5, the time on the perfect clock will be 6.2, which is larger than 6. Therefore, even if b arrives when the monitor is monitoring the interval, the monitor can still determine b does not belong to it. This example suggests that the deadline for monitoring an interval depends only on the duration of the interval and the relationship between the monitoring clock and the perfect clock, but not on the system clock. At the same time, to determine whether an event is within an interval depends on the relationship between the system clock and the perfect clock, but not on the monitor clock. We will make this intuition precise below.

6.2. System Architecture and preliminaries

In this section, we will present the architecture for monitoring time intervals. Then some preliminaries are given, including definitions of some key concepts and parameters of monitoring setup to be explored.

6.2.1. Architecture

Figure 22 illustrates the architecture for monitoring time intervals. To separate the logic of time management, a module $IntervalHandler$ is introduced into the monitor between the target system and the property checker. Both the IntervalHandler and checker run under the monitor clock. The checker can be implemented in SMEDL or other event-driven RV tools. It receives two types of events from the IntervalHandler, one is the original events for property evaluation. Another is a special event *interval closure* introduced above, which is used to acknowledge to the checker the end of a time interval.

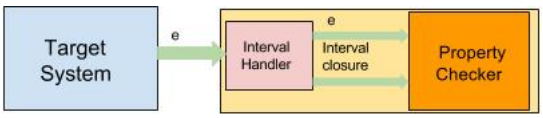

Figure 22: Architecture for monitoring time intervals

A checker correctly evaluates the property for a time interval i if all events occurring in i are delivered to the checker when the property is being evaluated. In the ideal situation, when events are delivered from the system to the monitor immediately and there is no timing uncertainty, this can be easily achieved by setting the timer in the IntervalHandler for the duration of i. Any event arriving before the timer expires would be within i, while any event arriving after it expires is outside i. Expiration of the timer immediately raises the closure event. If events can be delayed, however, this approach may clearly result in incorrect checking. The closure event must be delayed to accommodate for late events. In order to close the interval in a timely manner, we need to set a *deadline* for raising the closure event that would guarantee correct monitoring and minimize the delay in closing the interval. According to the duration on the time interval and parameters of the monitoring setup, the IntervalHandler calculates the deadline for each interval. When the current time at the monitor reaches the deadline, the IntervalHandler sends an interval closure event to the checker to finish the evaluation of the property for this interval.

The deadline discussed above is useful in another way. If events arrive out of order, they also should be re-ordered according to their timestamps before being passed on to the checker which, as we

discussed above, does not reason about time. In our approach, the IntervalHandler is storing events in a queue in the timestamp order and uses the same deadline to release events from the queue to the checker. We discuss event reordering further in Section 6.4.

6.2.2. Preliminaries

Time model. There are three time domains assumed: T_m for the monitor clock, T_s for the system clock and T_p for the perfect clock. The monitor takes streams of events as input. Events are timestamped using the *system clock* in the time domain T_s. The system and monitor clock may be skewed and run at different rates. In addition, there may be unpredictable delays in delivering events from the target system to the monitor. As a result, event timestamps are not directly comparable with readings of the monitor clock. Moreover, elements in the time domain T_m and T_s are totally ordered. An event stream E_T is a sequence of timestamped observations $\langle(o_1, t_1), (o_2, t_2), ...\rangle$, where o_i is a value observed at time $t_i \in T_s$. The perfect clock c_p in T_p is used to measure the length of the time interval being monitored.

Time interval is a period of time between two events, the duration of which is measured by the perfect clock. In the remainder of this chapter, when we refer the interval on the system, we use "start" and "end" to denote the beginning and ending of the interval. On the monitor side, an interval is "opened" or "closed" by the monitor. A closed interval i that starts at t_1 and ends at t_2 is denoted as $i_{[t_1,t_2]}$. For an event e originated from the system and an interval i, if $t_1 \leq t_e \leq t_2$, then $e \in i_{[t_1,t_2]}$ where t_e is the timestamp of e. Note that if we don't care about events occurring on the bound(s), the interval could also be half-open or open and the denotation will be modified accordingly.

Network delay, denoted as nd, represents the time to send the event from the system to the monitor. The absolute value of the delay is measured in the sense of perfect clock.

Clock rate is the interval of the finest time unit. It is assumed that the clock rate of c_p, denoted as r_p, is 1. The clock rate of the system and the monitor are respectively denoted as r_s and r_m. If r_s (r_m) is greater than 1, then the system (monitor) clock runs ahead of the perfect clock.

Clock skew, denoted as ts, represents the time difference $t_m - t_s$ between the monitor and the system where t_s is the time of the system and t_m is the time of the monitor. Here we assume that time synchronization is periodically conducted between (1) the system clock and the perfect clock and (2) the monitor clock and the perfect clock.

6.3. Setting the Interval Deadline

In this section, we explore the parameter space of network delay, clock skew and clock rate and identify several cases where correctness of monitoring can be ensured. For each case, we describe how to calculate the deadline for closing the interval. The monitor uses this deadline to set the timer; when the timer expires, we can be certain that no further events belonging to this interval can arrive and the closure event is sent to the checker. Patterns of setting the timer for non-recurrent and recurrent intervals are presented respectively. Case analysis on the three parameters is conducted.

6.3.1. Patterns of setting timer

We rely on timers to determine when an interval can be closed. The timers are set differently based on whether the interval is recurrent or non-recurrent, shown in Figure 23. Note that the clock rate of the system rs is used to calculate the actual time on the system side.

Non-recurrent intervals. Here we only consider the case involving two consecutive intervals such as the property $aU_{[t_1,t_2]}b$. In $aU_{[t_1,t_2]}b$, two intervals, $[0, t_1)$ and $[t_1, t_2]$, are involved. The monitor begins checking $[0, t_1)$ and $[t_1, t_2]$ when a arrives and two corresponding timers are set to close the intervals.

Recurrent intervals. As the number of intervals to be monitored is unbounded, only the timer for the first interval is set. Then every time an interval is closed, the timer for closing the next interval is set with a proper monitoring deadline. In the following section, we will denote the duration of the recurrent interval as d.

In order to set the deadline as accurate as possible, two steps have to be done. The first step is to estimate the time on the monitor side when an event e occurs on the system side, denoted as

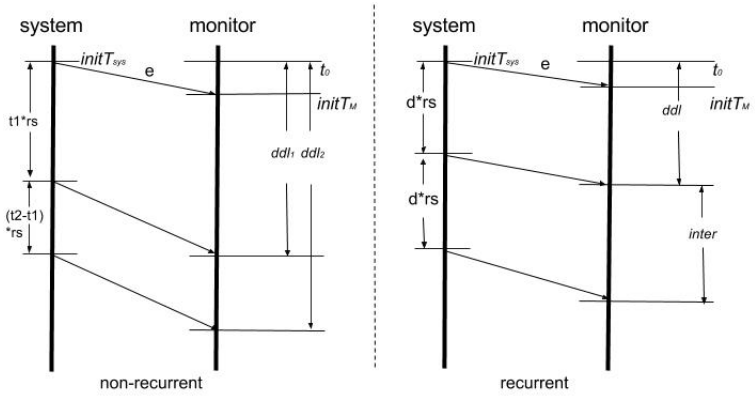

Figure 23: Scheme of setting deadlines for non-recurrent and recurrent intervals

t_0 in Figure 23. The second step is to calculate deadlines for each monitor based on t_0, which is introduced below.

6.3.2. Scheme of setting deadline

Here we give the case analysis with varying the values of the clock rate, network delay and clock skew with the assumption of bounded network delay. Figure 23 illustrates the scheme of setting deadline for non-recurrent and recurrent intervals. The time when the initial event e occurs on the system side is denoted as $initT_{sys}$, measured by the system clock and $initT_M$ is the time at the monitor when e arrives at the monitor.

The monitor begins the monitoring process at $initT_M$. For the non-recurrent case, ddl_1 and ddl_2 for interval $[0, t_1)$ and $[t_1, t_2]$ need to be calculated. Then, as the timers are set at $initT_M$ with a relative value, deadline for $[0, t_1)$ is set with value $ddl_1 - initT_M + t_0$ and deadline for $[t_1, t_2]$ is set with value $ddl_2 - initT_M + t_0$. For the recurrent case, the deadline for the first interval can be calculated in a similar way to the non-recurrent case: ddl is calculated according to the duration of interval and the monitoring setup and the deadline for the first interval is $ddl - initT_M + t_0$. From the second interval, timers are set with a period $inter$. The reason the first interval is different from the rest of them is that for the initial event e, we know the exact time when e arrives at the monitor, but for the rest of intervals, we only consider the worst case where the last event for a interval occurs at the boundary and the delay for the delivery is the maximum value of the network

delay. In the following case analysis, we will estimate the value of t_0 and calculate ddl_1 and ddl_2 for the non-recurrent case; ddl and $inter$ for the recurrent case.

Case 1 : $r_s = 1$, $r_m = 1$, $nd = 0$. In this case, interval durations of the system and monitor are identical and there is no delay, so $t_0 = initT_M$. For the case of non-recurrent intervals, ddl_1 and ddl_2 are respectively t_1 and t_2. For the case of recurrent intervals, ddl and $inter$ have the same value d since there is no network delay.

Case 2 : $r_s = 1$, $r_m = 1$, nd is fixed and known. In this case, clock skew ts can be directly calculated by $initT_M - initT_{sys} - nd$ and $t_0 = initT_{sys} + ts$. For the case of non-recurrent intervals, ddl_1 and ddl_2 are respectively $t_1 + nd$ and $t_2 + nd$ since events occurring at the boundary of these two intervals have the delay of nd. For the case of recurrent intervals, ddl is set to $d + nd$, similar to the case of the non-recurrent interval. The value of *inter* is set to d because the interval is of length d and the network delay has already been taken into consideration when calculating the deadline of the first interval.

Case 3 : $r_s = 1$, $r_m = 1$, $nd \in [b1, b2]$, ts is known. As ts is known, $t_0 = initT_{sys} + ts$. We only need to consider the worst case in which network delay has the maximum value, which is when an event e with timestamp t arrives on the monitor side at $t + b2$. The least delay $b1$ is not relevant for computing deadlines. For the case of non-recurrent intervals, ddl_1 and ddl_2 are respectively $t_1 + b2$ and $t_2 + b2$. For the case of recurrent intervals, ddl is $d + b2$ and $inter$ has value d.

Case 4 : $r_s = 1$, $r_m = 1$, $nd \in [b1, b2]$, ts is unknown. The analysis is similar to the case 3 but t_0 cannot be determined precisely since ts is unknown and network delay is not fixed. Consequently, we approximate its value using the network delay. The worst case is when the value of t_0 is as late as possible. Therefore, we set $t_0 = initT_M - b1$. The same formulas setting deadlines used in case 3 are also used here.

Case 5: r_s is fixed, $r_m \in [r3, r4]$, $nd \in [b1, b2]$, ts at time $initT_{sys}$ is known. Like in case 3, t_0 is calculated using the formula $t_0 = initT_{sys} + ts$. Because of the clock rate difference between the system and the monitor, clock skew may change. However, since we do not compare time values

Table 22: Summary of deadline setting scheme

Monitoring setup		Non-recurrent		Recurrent	
	t_0	ddl_1	ddl_2	ddl	$inter$
$r_s = 1$, $r_m = 1$, $nd = 0$	$initT_M$	t_1	t_2	d	d
$r_s = 1$, $r_m = 1$, nd is fixed and known, ts is known	$initT_{sys}+$ ts	$t_1 + nd$	$t_2 + nd$	$d + nd$	
$r_s = 1$, $r_m = 1$, $nd \in [b1, b2]$, ts is known	$initT_{sys}+$ ts	$t_1 + b2$	$t_2 + b2$	$d + b2$	
$r_s = 1$, $r_m = 1$, $nd \in [b1, b2]$, ts is unknown	$initT_M-$ $b1$				
r_s is fixed, $r_m \in [r3, r4]$, $nd \in [b1, b2]$, ts at time $initT_{sys}$ is known	$initT_{sys}+$ ts	$(t_1+b2)*$ $r4$	$(t_2+b2)*$ $r4$	$(d + b2) * r4$	$d * r4$
r_s is fixed, $r_m \in [r3, r4]$, $nd \in [b1, b2]$, ts is unknown	$initT_M-$ $b1 * r3$				

between the system and the monitor anywhere else, the value of the clock skew does not affect calculations of the deadline value. To cover the worst case of event arrival when calculating the deadline, r_m and nd need to be at their upper bounds. For the case of non-recurrent intervals, ddl_1 and ddl_2 are respectively $(t_1 + b2) * r4$ and $(t_2 + b2) * r4$. For the case recurrent intervals, ddl has value $(d + b2) * r4$ and $inter$ has value $d * r4$.

Case 6: r_s is fixed, $r_m \in [r3, r4]$, $nd \in [b1, b2]$, ts is unknown. Similar with case 4, we need to approximate t_0 using its maximum value: $initT_M - b1 * r3$. The formulas used in case 5 are used in this case.

One can observe that case 5 and 6 are generalization of special cases 1 to 4 and there is no conflicts between them. The summary of case analysis on deadline setting is shown in Table 22. We can prove that given monitoring setup in case 5 and 6, correctness of monitoring intervals can be guaranteed, shown in Lemma 42.

Lemma 42 (Correctness of Monitoring Interval for Setup in Case 5 and 6) *If r_s is fixed, r_m is fixed and known with in the range $[r3, r4]$ and $nd \in [b1, b2]$, we can always set a deadline for monitored intervals as illustrated in Table 22, such that all events of the interval will fall within the*

deadline.

Proof(sketch): Based on whether ts is known at the beginning of monitoring process, we split into two cases corresponding to case 5 and 6 above. Here we give the sketch for proving the case of monitoring non-recurrent intervals $[0, t_1)$. The proof for interval $[t_1, t_2]$ and recurrent intervals is similar. Recall that t_0 is the estimated time, by the monitor clock, when the initial event occurs on the system side. The deadline is set in two steps, illustrated in Figure 23, and we argue correctness of these two steps separately. First, we compute the largest possible value for t_0 and this is correct because 1) if ts known, we can calculate the accurate time t_0 of the monitor given the timestamp of $initT_{sys}$ when the initial event occurs on the system side; and 2) if ts is not known, we compute t_0 having the maximum value using the $initT_M$ and the lower bound of nd. Then, we set the deadline relative to t_0 and we do it correctly because we over-estimate the deadline with the upper bound of r_m and nd. We then compare the deadline with t_r, the relative time between $initT_{sys}$ and the latest possible arrival time of the event occurring at t_1 at the monitor. The value of t_r is $t_1 + b2$ in the sense of perfect clock. Translating deadline to the perfect time scale, the value would be $(t_1 + b2) * r4/r_m$, which is greater than or equal to t_r. Since t_0 is equal to or greater than the time when the initial event occurring within the interval, we can always ensure that all events will fall within the deadline. □

Lemma 42 can be extended to Theorem 43 describing sufficient condition for correctly monitoring time intervals.

Theorem 43 (Correctness of Monitoring Interval) *If r_s is fixed, r_m is fixed and known with in the range $[r3, r4]$ and nd is bounded, we can set a deadline for each monitored interval as illustrated in Table 22 such that all events of the interval will arrive at the monitor within the deadline.*

Proof(sketch): The proof proceeds by case analysis of entries in Table 22. Note that cases 1-4 are special cases of 5 and 6 and need not be considered separately. The union of the monitoring setup conditions in Table 22 is exactly the premise of the theorem. Therefore, correctness of cases 5 and 6, established by Lemma 42, proves the theorem. □

6.4. Monitoring Procedure

This section presents the procedure for monitoring time intervals using the scheme of setting monitoring deadline proposed in the previous section. The procedure describes operation of the IntervalHandler introduced in Section 6.2.1.

The procedure relies on two key functions. First, $calculateDeadline$ sets the deadline for each interval according to Section 6.3. Second, $getInterval$ is given an event and returns an interval to which this event belongs, as follows. Given an event e with the timestamp t and $initT_{sys}$ which indicates the occurring time of the initial event, we need to get the interval that e belongs to. With the condition that the rate of the system r_s is fixed, the interval can be determined. For the non-recurrent interval, if $t - initT_{sys} < t_1 * r_s$, e belongs to the interval $[0, t_1)$; if $t_1 * r_s \leq t - initT_{sys} \leq t_2 * r_s$, e belongs to the interval $[t_1, t_2]$; otherwise, e falls out of these two intervals. For the recurrent interval, the interval is calculated using the formula $\lfloor (t - initT_{sys})/(d * r_s) \rfloor$. As stated below, each interval is identified by an integer according to the order, starting from 0.

Figure 24 shows the detailed structure of the $IntervalHandler$ and how it connects to the $Property\ Checker$. The IntervalHandler is responsible for managing intervals and the checker evaluates the logic of the property. Note that the monitoring process is slightly different between the cases of in-order-delivery and out-of-order delivery. The $IntervalList$ is the data structure representing intervals of interest. In the non-recurrent case, there are the two intervals $[0, t_1)$ and $[t_1, t_2]$. In the recurrent case, if in-order delivery is assumed, we just need to remember the earliest non-closed interval. For out-of-order delivery, the IntervalList needs to remember all non-closed intervals for which at least one event has been received. We also associate a data structure $eventQueue(i)$ for each interval i in the IntervalList: each arrived event is put into the corresponding eventQueue ordered by the timestamp. Once the interval i is closed — that is, no more events from this interval can arrive, — the IntervalHandler sends all events in the eventQueue(i) to the checker, followed by the interval closure event.

In the IntervalHandler, $intervalManager$ is used to relay events from the system and manage in-

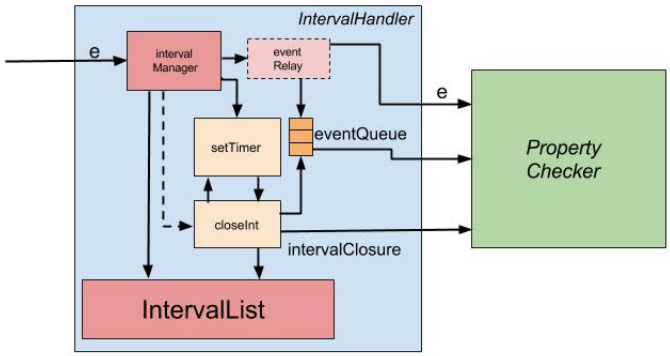

Figure 24: Structure for the IntervalHandler

tervals. It first examines whether the received event e is the initial event arriving at the monitor. If so, it computes the deadline and sets the timer for the first interval. According to the setting of network delay, clock rate and clock skew, calculateDeadline computes the deadline using the value of $initialTS$, $initialTM$ as well as the left and right boundary of the interval. The second attribute of $setTimer$ is the index of the corresponding interval upon which the timer is set. In the case of properties involving two non-recurrent intervals, two timers with corresponding deadlines need to be set. Then, the interval i that e belongs to is computed. If in-order delivery is assumed, the current interval being evaluated by the checker, denoted as i', is obtained from the IntervalList by calling the procedure $getLeastOpenedInt$. If i is not equal to i', i' is closed and the corresponding timer will also be unset. Event e is then sent to the checker. If out-of-order delivery is assumed, it is put into corresponding $eventQueue(i)$.

```
void intervalManager (){
   while(true) {
      Interval i;
      Event e = receiveEvent();
      if (initialEvent(e)){
         initialTS = e.getSystemTimeStamp();
         initialTM = getcurrentTime();
         deadline = calculateDeadline();
         setTimer(deadline,0);
      }
      i = getInterval(e);
      if (out-of-order-delivery){
```

```
            addQueue(e,eventQueue(i));
        }else{
            i' = getLeastOpenedInt();
            if(i != i'){
                closeInt(i');
                unsetTimer(i');
            }
            PropertyChecker.handlingEvents([e]);
        }
}
```

Procedure $closeInt(i)$ is responsible for closing the interval i, which is called when the corresponding timer is up or an event for the next interval has arrived in the case of in-order delivery. It first calculates the deadline for the next interval $i + 1$ to be evaluated and sets the corresponding timer. For the case of non-recurrent interval, the timer is not reset. Then events in eventQueue(i) are sent to the checker if out-of-order delivery is assumed. Finally, $intervalClosure(i)$ is called to close the interval i and modify the IntervalList. For the case of recurrent interval, interval i is removed from the IntervalList and $i + 1$ is set as the earliest non-closed interval if in-order delivery is assumed.

```
void closeInt(integer i){
     ddl = calculateDeadline();
     setTimer(ddl, i+1);
     if (out-of-order-delivery){
       liste = getEventsForQueue(eventQueue(i));
       PropertyChecker.handlingEvents(liste);
     }
     intervalClosure(i);
 }
```

6.5. Summary

This section presented an approach to monitoring of time intervals in an event-driven fashion. To do this, we introduced an interval closure event, with the property that all events that fall into the interval occur before the interval closure. The two challenges are (1) correctness of the procedure and (2) timeliness of the event closure. To address these two challenges, we offered a procedure to determine when all events that can fit into the interval have been observed. The answer to this

question depends on parameters of monitoring setup, namely network delay, clock skew between the system and the monitor and clock rates of the two. We performed case analysis and show how to close intervals in different cases.

CHAPTER 7 : Reflexive Adaptation Framework

Runtime enforcement (RE) is a dynamic technique to guarantee satisfaction of formally specified properties in event traces. EMs (enforcement monitors), synthesized from these specifications, describe how to transform event streams using abstract adaptation actions (also referred to as enforcement actions) to preserve properties. During execution, EMs intercept actions generated from the target system and change the state of the program for adaptation. Although pre-defined adaptation actions are usually trivial and would not permanently solve the problems, they can be used as temporary repairs and are more feasible for validation compared to more sophisticated methods. In this chapter, we will extend SMEDL as an adaptation framework. Then, we present a method to statically verify correctness of actions with respect to the execution of the target program.

7.1. Extension of SMEDL framework for software adaptation

The architecture of the extended framework is illustrated in Figure 25. In the setting of online monitoring, program behaviors are extracted as events and delivered to monitors. For software adaptation, event delivery is bi-directional: events raised from monitors are transformed into executable adaptation actions, which are inserted at specified program points during runtime. In Section 2.3.3, we have briefly introduced several formalisms for runtime enforcement, which have different capabilities to transform input event streams. Among those formalisms, edit automata [96] has been successfully applied to enforce properties in Java [24] and Android applications [117]. Moreover, composition of multiple EMs has been studied. However, there is little work on analyzing interference between the adaptation actions and the target program. In this section, we encode edit automata using SMEDL. Furthermore, we present an initial work on it by defining an operational semantics to describe how the target program behaves after the integration of adaptation actions. We then further propose a method to generate the implementation of adaptation actions and instrument them into the target program.

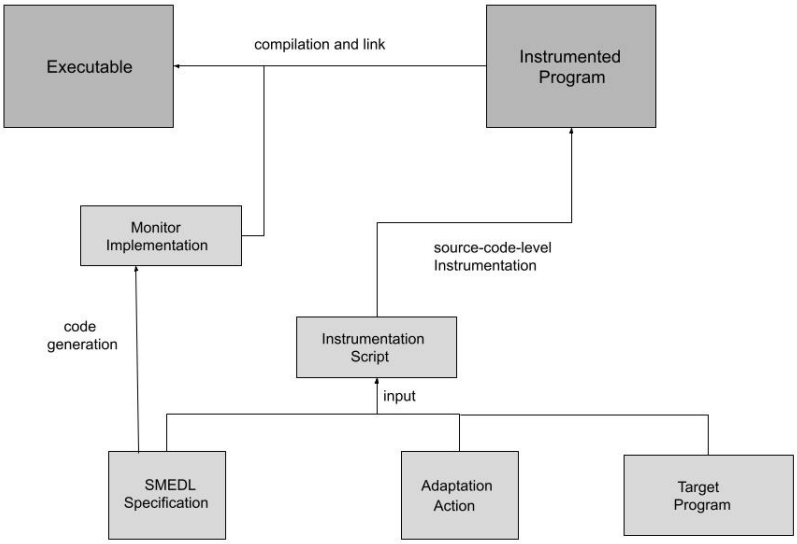

Figure 25: Architecture of the response framework

7.1.1. Introduction to edit automata

Figure 26 illustrates the automaton of the policy *Iterator_hasNext*, which says that *next* needs to be called when *hasNext* returns true and *next* cannot be called when the corresponding collection object has been modified. If a program breaks this policy by calling *next* without checking accessibility of the memory cell to be traversed, it may abort abnormally due to undefined behaviors. To enforce this specification, one can either *insert* a call to *hasNext* or *suppress* the call to *next* if *hasNext* has not been called, which can be described using edit automata.

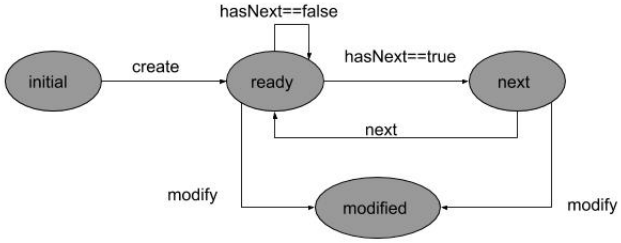

Figure 26: Policy of the Iterator as a state machine

Figure 27 illustrates two edit automata as the EMs to enforce the policy in Figure 26. When the iterator is created, each EM is initialized to the *ready* state. Each transition is decorated with a pair

of input event e and output event el, denoted e/el, which means that whenever e is received, the automaton takes the transition and outputs el. The event $hasNextTrue$ and $hasNextFalse$ respectively represent that the call to $hasNext$ returns true and false. The event $checkedNext$ represents conditional execution of $next$ when $hasNext$ returns true. Figure 27 (a) suppresses every call to *next* in the *ready* state while Figure 27 (b) inserts a call to *hasNext* and executes *next* when it returns true. Moreover, both EMs suppress all $next$ calls after the corresponding collection has been modified.

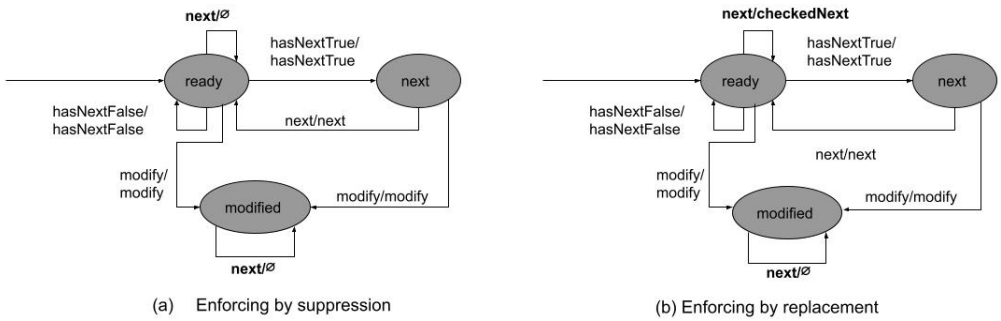

Figure 27: Two enforcement specification for the policy in Figure 26

Definition 44 (Edit automaton) *An edit automaton is a tuple (A, Q, q_0, δ) where A is an action set; Q is a set of states; q_0 is an initial state; δ is a partial function $Q \times A \rightharpoonup Q \times A^*$ that transforms an input action into a list of actions to emit and updates the state:*

- $(q, a) \rightarrow (q', a)$ *(nop)*
- $(q, a) \rightarrow (q', \cdot)$ *(suppression)*
- $(q, a) \rightarrow (q', \sigma; a)$ *(insertion)*
- $(q, a) \rightarrow (q', a')$ *(replacement)*

7.1.2. Encoding of edit automata

Edit automata can be encoded in a SMEDL monitor M monitor by defining a partial function $adapation\_mapping : M.E \rightharpoonup AdapationActions$ that maps events to a set of $adaptation$ $Actions$ where

$$adaptationAction := insert(action\_block) | suppress(n) | replace(n, action\_block)$$

The types of $adaptationAction$ correspond to the actions of edit automata. $action\_block$ contains a list of statements of general programming languages. The parameter n in $suppress$ *resp.* $replace$ indicates that the next n statements will be skipped *resp.* replaced with the code specified in the action block.

Operational semantics of adaptation actions. Since adaptation actions may change the state of the target program, we need to formally describe their behavior in the target program. First, we define an *action-step* to be applied after the macro-step is taken to obtain the actions to be executed:

$$\frac{conf \xrightarrow{e} conf' \quad \forall re' \in EX_{conf}, re'.event \notin dom(adaptation\_mapping)}{conf \hookrightarrow_e (conf', \cdot)} \quad (1)$$

$$\frac{conf \xrightarrow{e} conf' \quad re \in EX_{conf'} \wedge adaptation\_mapping(re.event) = action \quad \forall re' \in EX_{conf} \wedge re' \neq re, re'.event \neq re.event \wedge re'.event \notin dom(adaptation\_mapping)}{conf \hookrightarrow_e (conf', action)} \quad (2)$$

Rule (1) states the case in which there is no raised event that can map to an adaptation action. Rule (2) restricts that there must only exist one event that can trigger an adaptation action. It is straightforward to extend the rules of synchronous set to maintain this restriction. Combination of multiple adaptation actions is left as future work.

To describe how generated actions influence the execution of the target program, we define a set of semantic rules for the target program with respect to the monitor execution. Here are some assumptions on the setting: 1) the monitor has been synchronously instrumented to the target program; 2) the monitor execution is abstracted by calling $call\_monitor$ and the process of extracting the event e has been omitted; 3) instrumentation of adaptation actions will not cause compilation errors. For simplicity, our work does not support `abort` and `goto` and the monitor is not called in the adaptation actions.

We overload the operator $\Downarrow$ to represent the evaluation rule. The program state is an tuple $(\delta, conf)$ where δ is the store of the program state while $conf$ is the monitor configuration. The evaluation relation $\langle C, \delta, conf\rangle \Downarrow (\delta', conf')$ says that starting in configuration $(\delta, conf)$, the program is evaluated to $(\delta', conf')$ after executing the program statement C.

Since the program state and monitor state are disjoint, the monitor execution will not interfere with the target program if no adaptation actions are generated, as shown in the rule *nop*. The rule *insertion* states that τ, the code block of the adaptation action block generated by the monitor execution, will be executed at the instrumentation point. The rule *suppression* states that the next n statements after the instrumentation points will be skipped. The rule *replacement* suppresses the next n statements and then execute τ.

$$\frac{conf \hookrightarrow_e (conf', \cdot)}{\langle call\_monitor(e), \delta, conf\rangle \Downarrow (\delta, conf')}(nop)$$

$$\frac{\begin{array}{c} conf \hookrightarrow_e (conf', insert(\tau)) \\ \langle \tau, \delta, conf'\rangle \Downarrow (\delta', conf') \end{array}}{\langle call\_monitor(e), \delta, conf\rangle \Downarrow (\delta', conf')}(insertion)$$

$$\frac{conf \hookrightarrow_e (conf', suppress(n))}{\langle call\_monitor(e); stmt_1; ...; stmt_n, \delta, conf\rangle \Downarrow (\delta, conf')}(suppression)$$

$$\frac{\begin{array}{c} conf \hookrightarrow_e (conf', replace(n, \tau)) \\ \langle \tau, \delta, conf'\rangle \Downarrow (\delta', conf') \end{array}}{\langle call\_monitor(e); stmt_1; ...; stmt_n, \delta, conf\rangle \Downarrow (\delta', conf')}(replacement)$$

7.1.3. Code generation of adaptation actions

In our current setting, adaptation actions are always executed at instrumentation points where the monitor API is called and the code to be suppressed or replaced is statically determined. This section presents a procedure that generates the adaptation code that are instrumented into the target system. Note that the code to be constructed is C-like pseudo code, which can be easily transformed into

executed code.

Given an imported event, there may be multiple possibilities of which action block will be executed and the restriction in the semantics guarantees that at most one event that maps to adaptation actions is raised. Algorithm 3 computes a *code template* for each imported event. For the input parameters, mon is a monitor specification; $action\_map$ is an $adaptation\_mapping$; $call\_monitors$ is a map from imported events to corresponding monitor APIs. The algorithm traverses all imported events defined in mon (line 3). The procedure $computeDependency$ (line 6) computes a set of exported events that may be raised by transitions triggered directly or indirectly by e. The corresponding action block can be obtained from $action\_map$ (line 8). The variable max is used to store the largest code block that may be suppressed or replaced (line 9-10). Action blocks are added to $action\_lst$ (line 11). The implementation of $call\_monitor\_e$, the monitor API for e, is updated to add return value (line 13). As presented below, the return value decides which action block to execute. The procedure $buildReturnVal$ constructs the code to generate the return value, which is the index of the corresponding action in $action\_lst$ that the exported event maps to. For each imported event e, the code template is constructed (line 14).

Algorithm 3 Construction of template for adaptation actions

```
1:  procedure CONSTRUCTTEMPLATE(mon, action_map, call_monitors)
2:      switch_map ← ∅
3:      for e ∈ mon.ImportedEvents do
4:          max ← 0
5:          action_lst ← ∅
6:          exported_evset ← computeDependency(mon, e)
7:          for ev ∈ exported_evset do
8:              action ← action_map(ev)
9:              if (action = suppress(n)||action = replace(n, _)) && max < n then
10:                 max ← n
11:             action_lst ← action_lst.append(action)
12:         call_monitor_e ← call_monitors(e)
13:         call_monitor_e ← call_monitor_e.code.append(buildReturnVal(action_lst))
14:         codeTemplate ← ⟨call_monitor_e, max, action_lst⟩
15:         switch_map ← switch_map ∪ [e ↦ codeTemplate]
16:
        return switch_map
```

After computing the code template for each imported event, Algorithm 4 generates the code for

each instrumentation point in the program. Note that instrumentation of the monitor API is omitted. For the input parameters, $prog$ is the target program to be instrumented; $switch\_map$ is the code template generated by Algorithm 3; and $instru\_map$ is the map from the instrumentation point (represented as a line number in the program) to the corresponding imported event extracted at it. The algorithm proceeds by traversing all instrumentation points (line 2). At line 4, the code block (denoted as $code$) to be replaced by the code block $switch$ is obtained. $c.subList(n1, n2)$ returns a list of statements of c from line *n1* (included) to *n2* (excluded). As its name indicated, $switch$ is a code block in the form of a switch structure. The guard condition of $switch$ is the return value of $call\_monitor\_e$ (line 5), which is decided by the event raised by the monitor. Then, for each action that may be triggered by e, a corresponding case is constructed (line 7-15). For insertion (line 9), the action τ is inserted before $code$. For suppression, the first n statements of $code$ are removed (line 11). For replacement, the first n statements of $code$ are replaced by τ (line 13). Note that n is smaller or equal to max. The condition for each case is the index value (idx) in the $action\_lst$ of the template, which is consistent with the return value of the monitor API constructed in Algorithm 3. After $switch$ is constructed, it replaces $code$ in $prog$ (line 16). Note that the position is p + 1 because the monitor API would be instrumented at p.

Algorithm 4 Instrumentation of the adaptation implementation into the target program

```
1:  procedure INSTRUMENTACTIONS(prog, switch_map, instru_map)
2:      for (p, e) ∈ instru_map do
3:          codeTemplate ← switch_map(e)
4:          code ← prog.subList(p, p + codeTemplate.max)
5:          switch.guard ← codeTemplate.call_monitor_e
6:          idx ← 0
7:          for action ∈ codeTemplate.action_lst do
8:              if action = insert(τ) then
9:                  new_code ← τ.append(code)
10:             else if action = suppress(n) then
11:                 new_code ← prog.subList(p + n, p + codeTemplate.max)
12:             else if action = replace(n, τ) then
13:                 new_code ← τ.append(prog.subList(p + n, p + codeTemplate.max))
14:             switch.cases ← switch.cases.append(idx, new_code)
15:             idx ← idx + 1
16:         prog.remove(p, code).insert(p + 1, switch)
        return prog
```

Figure 28: SMEDL specification for the edit automata in Figure 27 (b)

```
object enforceIterator;
    events:
        imported next();
        imported hasNextTrue();
        imported hasNextFalse();
        imported modify();
        exported suppress();
        exported checkedNext();
    scenarios:
       main:
                  ready -> next() {raise checkedNext();} -> ready;
                  ready -> hasNextTrue() -> next;
                  next -> next() -> ready;
                  next -> modify() -> modified;
                  ready -> modify() -> modified;
                  modified -> next() {raised suppress();} -> modified;
```

Figure 29: $adaptation\_mapping$ for *iterator-enforcer*

```
  suppress => suppress(1);
  checkedNext => replace(1, checkNextBlock);
  checkNextBlock{
        if(hasNext(&i)){
            value = next(&i);
        }
  }
```

We use the iterator policy in Figure 26 to illustrate how to specify an adaptation specification and generate the code to be inserted in the target program. Figure 28 is the SMEDL specification for the edit automata in Figure 27 (b) (denoted as *iterator-enforcer*). The exported event $checkedNext$ and $suppress$ correspond to the adaptation action triggered at the state $modified$ and $next$. The $adaptation\_mapping$ and the action block are shown in Figure 29.

Figure 30 is a code block that uses the iterator API. By analyzing *iterator-enforcer*, we know that there are two adaptation actions that may be triggered by the API *next*. As stated above, return value of the monitor call decides which action to execute. The monitor API, $call\_monitor\_next$, may return two values, 0 and 1, corresponding to $suppress$ and $checkedNext$ in *iterator-enforcer*. The code is instrumented into the program as shown in Figure 31. Note that although *checkNextBlock* is not parameterized with the iterator, it can be achieved by connecting the parameter of the monitor to the parameter in the action block. We will leave it as future work.

Figure 30: Original code block

```
...
int value = 0;
while(k < n){
  value = next(&i);
  if(value == v) {
      idx = i.iterator_pointer-1;
  }
 k++;
```

Figure 31: Instrumentation of the adaptation action

```
...
int value = 0;
while(k < n){
  int ret = call_monitor_next(&i);
  switch(ret){
     case 0: break;
     case 1:
         if(hasNext(&i)){
            value = next(&i);
         }
         break;
   }
  if (value == v) {
      idx = i.iterator_pointer-1;
  }
 k++;
}
```

As mentioned in Section 2.3.3, correctness is an important issue for runtime adaptation. At the specification level, soundness and transparency are two principles [96], which has been well-studied [96, 52, 27, 83, 26]. At the level of implementation, existing works have used type-theory [69], theorem proving [4] and model checking [129] to guarantee that the implementation of monitor code follows the semantics of the specification. In Chapter 3, we have also proposed a correct-by-construction code generation of monitor code from Coq. However, there is little work, to the best of our knowledge, on analyzing how a poor implementation of an action or ill-formed instrumentation may interfere with the application. The next section presents an approach to solve this problem.

7.2. Verification of adaptation actions

To further motivate this problem, we consider a use case of an application that uses a library API. Figure 32 illustrates assume-guarantee reasoning to verify correctness of an application that uses an external library through a well-defined API. Such a correctness proof contains two parts. The first part, performed on the application itself, proves functional correctness of the application with respect to given API specifications. The second part, performed on the library, is to prove correctness of API specifications. Modular verification [78] can usually tie these two parts together if the pre-

condition of the API can be easily checked at each call site. However, for many APIs, correctness of the specification depends on whether the application adheres to a behavioral policy over sequences of API calls, e.g. in the form of an automaton such as the one shown in Figure 26 that illustrates the policy of using iterator.

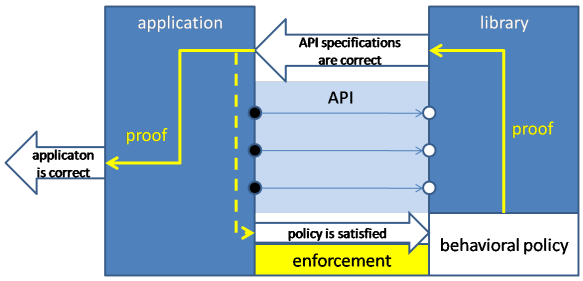

Figure 32: Overview of the assume-guarantee framework and our approach to assurance

Instead of statically proving compliance of the policy, which requires sophisticated interactions between the application and the library, users can use runtime enforcement to enforce the policy. In this way, they only need to focus on proving correctness of the application and the synthesis and integration of EMs can be fully automated. However, a poorly implemented action may interfere with the application and do harm, e.g. infinite loop due to ill-formed instrumentation.

To analyze the influence of actions, we define correctness of enforcement actions with respect to the specification of the original application, which means that, *given the application code instrumented with the actions, we can still construct the proof for the same specification*. We assume that functional correctness of the program is described as pre- and post-conditions and proved by generating verification conditions using weakest precondition calculus. These verification conditions (also denoted as proof obligations) are discharged by theorem provers or proof assistants. When analyzing correctness of the enforcement action, a straightforward way is to reconstruct the proof obligation globally using a theorem prover. However, this method cannot be easily scaled up to multiple instrumentation points because it assumes that any enforcement action may influence the program globally. Each time a new instrumentation point is added to the program, proof must be reconstructed globally. We consider a way of incrementally updating the proof of the original application to take into account effects of instrumentation, reusing most of proof obligations in the

original proof. Proof at each instrumentation point can then be constructed independently without influencing other parts of the program. We introduce a method for such an incremental update of the proof and specify restrictions that make such an update possible.

Motivating example. We use a variant of the *find* algorithm in the C++ Standard Library [127] which implements sequential search for an array. Instead of searching through the array directly, the iterator is used, as shown in Figure 33. The behavioral specification language ACSL (ANSI/ISO C Specification Language) [20] is used to describe the property of the program. The program uses the iterator *i1* to traverse the target array and checks whether the value *v* is in that array. A flag array is traversed by *i2*. When the value of the cell in the flag array (denoted as *b* in the program) is not zero, the corresponding cell in the target array can be accessed. The parameter *n* is the upper bound of the number of elements to be searched. The loop invariant states that if *idx* is not -1, the corresponding cell in the array is equal to *v*. The loop also has a variant specifying the decrement of an integer expression at each loop execution to guarantee termination. The specification of *next* (which is given in Section 7.2.6) guarantees that the call to *next* always returns the next value in the array given that the iterator has not reached the last element. Figure 34 and Figure 35 respectively implement the adaptation actions in Figure 27 (a) and (b). To save space, only the first call to *next* is enforced. We adopt the pattern proposed in Chapter 7.1.3 but instead of using the *switch* structure, we use the `ITE` (if-then-else) structure. If the return value is 1, enforcement actions will be executed; otherwise, the program code is executed as before. The suppression implementation in Figure 34 mistakenly includes the increment of i in the `else` branch, which breaks the loop variant and makes the program unable to terminate. Figure 35 gives a correct implementation.

Our method aims at detecting interference of adaptation implementations. It is worth noting that there may be no universally correct implementation, and if the program was implemented differently, a different implementation may be correct. We assume that adaptation implementations to be verified have been instrumented to the target program at a specified program point.

Figure 33: Original code of *find*

```
int find(iterator i1,
   iterator i2, int n, int v)
{
int i = 0;
int idx = -1;
/*@
  loop invariant idx != -1
  ==>i1.iterator_c.array[idx]==v;
   loop variant n-i;
*/
while(i < n){
  L1:
  int b = next(&i2);
  if(b){
    L2:
    int value = next(&i1);
    if (value == v) {
      idx = i1.iterator_pointer-1;
    }
  }
   i++;
}
  return idx;
}
```

Figure 34: Suppression

```
int find(iterator i1,
   iterator i2, int n, int v)
{
int i = 0;
int idx = -1;
/*@
  loop invariant idx != -1
  ==>i1.iterator_c.array[idx]==v;
 loop variant n-i;
*/
while(i < n){
     L1:
     int b = 0;
     if(call_monitor_next(&i2)){
          //suppression
     }else{
         b = next(&i2);
         if(b){
           L2:
           int value = next(&i1);
           if (value == v) {
             idx = i1.
             iterator_pointer-1;
          }
         }
         i++;
     }
  }
  return idx;
}
%\end{figure}
```

Figure 35: Insertion

```
int find(iterator i1,
   iterator i2, int n, int v)
{
int i = 0;
int idx = -1;
/*@
  loop invariant idx != -1
  ==>i1.iterator_c.array[idx]==v;
 loop variant n-i;
*/
 while(i < n){
  L1:
     int b = 0;
     if(call_monitor_next(&i2)){
       if(hasNext(i2))
          b = next(&i2);
       else
          b = 0;
     }else{
       b = next(&i2);
     }
     if(b){
       L2:
           int value = next(&i1);
           if (value == v) {
             idx = i1.
             iterator_pointer-1;
          }
     }
     i++;
 }
 return idx;
}
```

7.2.1. Target program and annotation language

We consider the program being verified and enforced to be a function written in a subset of C language. The program starts with a set of variable definitions with initialization. Variables can be either primary or struct variables. Statements include simple assignments, `ITE`, `while-loop` and `return`. `goto`, `break` or `continue` are not supported. There is only one exit point for the program. The right-hand-side (rhs) of the assignment can be normal arithmetic or logical expressions or a function call. All terms in the expression are either literal values or variables. Enforcement actions are implemented either as program instructions to be inserted or control structure to suppress execution of existing instructions in the application. To avoid complexity when applying the weakest precondition calculus, actions must not free existing heap space or change the value of existing pointer variables. Furthermore, auxiliary variables defined for the enforcement actions must only be accessed locally.

We use a subset of ACSL to annotate the target program and specify properties. The grammar is

given below. Annotations can be added as function contracts, assertions or loop invariants. The function contract is decorated as a set of pre- and post-conditions. Assertions are inserted before any C statement or at the end of blocks. Loop invariants are added at the head of the while loop. Each loop can also have a loop variant with an integer term that must be decreased in each loop execution. The *assigns* clause specifies memory locations that can be modified by the function.

⟨*function-contract*⟩ ::= ⟨*requires-clause*⟩* ⟨*simple-clause*⟩*

⟨*requires-clause*⟩ ::= '`requires`' ⟨*predicate*⟩ '`;`'

⟨*simple-clause*⟩ ::= ⟨*assigns-clause*⟩ | ⟨*ensures-clause*⟩

⟨*assigns-clause*⟩ ::= '`assigns`' ⟨*locations*⟩ '`;`'

⟨*ensures-clause*⟩ ::= '`ensures`' ⟨*predicate*⟩ '`;`'

⟨*locations*⟩ ::= ⟨*location*⟩ (, ⟨*location*⟩)* | \nothing

⟨*assertion*⟩ ::= '`assert`' ⟨*predicate*⟩ '`;`'

⟨*loop-annot*⟩ ::= ⟨*loop-invariant*⟩* $\langle loop - variant \rangle^{?}$

⟨*loop-invariant*⟩ ::= '`loop invariant`' ⟨*predicate*⟩ ;

⟨*loop-variant*⟩ ::= '`loop variant`' ⟨*term*⟩;

7.2.2. Weakest precondition calculus in Frama-C

We use Hoare logic [77] and the weakest precondition (WP) calculus to construct the proof. The implementation of the WP calculus in Frama-C is presented to fluently transition to the subsequent section for the methodology. We implement the verification method in the Frama-C (FRAmework for Modular Analysis of C code) tool developed by the CEA LIST and Inria. The WP plugin of Frama-C performs the weakest precondition calculus to transform properties into a set of first-order logic formula as proof obligations, which can then be discharged either using SMT solvers such as Alt-ergo [43], Z3 [45] or interactive proof assistants such as Coq [25].

Although the method we are proposing is based on general Hoare logic, some technical details in the design of the algorithm are specific to the implementation of WP calculus in Frama-C. The

WP plugin computes the verification condition by traversing the control flow graph (CFG) of the program and applying the WP calculus from the post condition of the program. A CFG of a function is a 4-tuple $\langle G, E, S, T \rangle$ where G is a set of nodes representing program elements; E is a set of edges representing the transfer of control; S is the start node and T is the end node. Multiple types of nodes are defined according to the corresponding program elements. Annotations are attached to edges. For each function, the WP plugin traverses the CFG backwardly from the end node to the start node and applies the WP rules to generate verification conditions for each node and edge. The verification condition at each node (edge) is computed by first obtaining the verification condition from its adjacent edges (node) and updating it with the information at the current node or edge. During traversal, each assignment is transformed into an SSA (single static assignment) before applying the WP rule. Therefore, predicates in the verification condition use *variable instances*. A variable instance takes the form var_idx where var is a variable and idx is its integer index. Whenever a variable is updated in an assignment, a new variable instance is created by incrementing the index. Multiple variable instances and the SSA transformation introduce technical challenges to our approach, which will be addressed in Section 7.2.5.

The verification condition is defined as a tuple $\langle \Sigma, VCS \rangle$ where Σ maps variables to variable instances to obtain the next variable instance when transforming into SSA. Moreover, at each program point, each variable instance in Σ also represents the state of that variable. We will use σ to refer an instance of Σ. *VCS* is a set of sub-obligations. The post condition of a function contract, assertions, the precondition of function calls and loop invariants are added as sub-obligations during traversal of the CFG. When applying the WP rules, replacement of variables and update of logic formula are performed by adding hypotheses to the post condition. Thus, each sub-obligation is represented as a tuple $\langle hyps, goal \rangle$ where $hyps$ and $goal$ are respectively the hypothesis and the proof goal.

7.2.3. Formal description of our approach

We study the program that uses API calls and its correctness has been proved in the Hoare proof system given that the API specification is validated. Enforcement actions are instrumented in the program to guarantee satisfaction of the API policy. We then need to prove that the instrumented

enforcement actions do not break the proof of the original program. The core idea is that if the instrumented enforcement actions influence the program only locally, the original proof is still valid when the implication relation between the verification condition of the original and modified program at the instrumentation point can be proved.

For notation, the original program is denoted as F and the instrumented program is denoted as F'. Enforcement actions are instrumented at the program point s. We assume the enforcement action to be instrumented is insertion of a series of actions *acts* at s. For suppression or replacement of instructions, the only difference is how to obtain the verification condition for F and F'. The correctness property of the program is specified by a pre-condition (denoted as pre) and a post-condition (denoted as $post$).

We assume that the correctness criteria of F, which is defined as the proof obligation $pre \implies WP(F, post)$, has been proved. For F', instead of generating the proof obligation $pre \implies WP(F', post)$ globally, the method we are taking is to extract the proof obligation VC and VC' for F and F' at s. If the proof obligation $VC \implies VC'$ can be discharged, the instrumented actions do not break the correctness of the program which has been justified by the original proof. To prove this statement, the following lemma is defined.

Lemma 45 *Given a block statement S and two verification conditions VC and VC' such that* $VC \implies VC'$*, then* $WP(S, VC) \implies WP(S, VC')$.

Proof (sketch): trivially by structural induction on S □

Theorem 46 *Assume programs F and F', F' is F with actions added at some point s and that* $pre \implies WP(F, post)$*. If there exist some verification conditions VC, VC' at point s in F and F' (respectively) such that* $VC \implies VC'$*, then* $pre \implies WP(F', post)$.

Proof (sketch): assume $pre \implies WP(F, post)$. Proof proceeds by structural induction on F at s. We present three representative cases; the others follow directly.

Case 1 (Block Statement): the instrumentation point s is located between block statements $S1$ and

$S2$, such that $pre \implies WP(F, post)$ is equal to $pre \implies WP(S1, WP(S2, post))$. We take VC to be $WP(S2, post)$ so $pre \implies WP(F, post)$ is equal to $pre \implies WP(S1, VC)$. Next, note that F' is $S1; acts; S2$, and so $pre \implies WP(F', post)$ is equal to $pre \implies WP(S1, WP(acts, WP(S2, post)))$. We take VC' to be $WP(acts, WP(S2, post))$. Then, if $VC \implies VC'$, by Lemma 45, $WP(S1, VC) \implies WP(S1, VC')$, and thus we can conclude $pre \implies WP(S1, VC')$, which is precisely $pre \implies WP(F', post)$.

Case 2 (`ITE` Statement): assume F takes the form $S1;\ If(b)\ \{S2; S3\}\ else\ \{S4; S5\};$. $WP(F, post)$ can be transformed into $WP(S1, b \implies (WP(S2; S3, post)) \wedge \neg b \implies (WP(S4; S5, post))$. Suppose for F', enforcement actions $acts$ are inserted between $S2$ and $S3$. $WP(F', post)$ can be transformed into $WP(S1, b \implies (WP(S2; acts; S3, post)) \wedge \neg b \implies (WP(S4; S5, post))$. We take VC and VC' to be $WP(S3, post)$ and $WP(acts, WP(S3, post))$. If $VC \implies VC'$, $WP(S2, WP(S3, post)) \implies WP(S2, WP(acts; S3, post))$ holds because of Lemma 45, which means $WP(F, post) \implies WP(F', post)$ holds. Since $pre \implies WP(F, post)$, $pre \implies WP(F',$
$post)$.

Case 3 (Loop Statement): the instrumentation is performed in the while loop. Assume F takes the form $S1; while(b)\{S2; S3\}$ and s is located between $S2$ and $S3$. Similar to the previous case, we need to prove that $WP(while(b)\{S2; S3\}, post) \implies WP(while(b)\{S2; acts; S3\}, post)$. According to the rule for the while loop, we only need to consider the case of invariant preservation, which can be reduced to case 2. □

Theorem 46 proves correctness of using local implication at one single instrumentation point. Since the construction of implication uses only the local information, the method is insensitive to the number of instrumentation points, as justified by Lemma 47 and Theorem 48.

Lemma 47 *Assume program blocks C and C', where C' is C with actions added at n program points, $s_1, s_2, ..., s_n$. If there exist some verification conditions $VC_1, VC'_1, ..., VC_n, VC'_n$ respectively at point $s_1, s_2, ..., s_n$ in C and C' such that $VC_k \implies VC'_k$ for all k from 1 to n, then*

$WP(C, post) \implies WP(C', post)$ *where post is the post condition of C.*

Proof (sketch): Without loss of generality, the program block C takes the form $C1; S1; S2; C2$ where $C1$, $S1$, $S2$, $C2$ are block statements. Denote $post$ as the post condition of C. After insertion of actions, we obtain C' as $C1'; act1; S1; S2; act2; C2'$ where $act1$ and $act2$ are enforcement actions and $C1'$ and $C2'$ are also modified from $C1$ and $C2$ by instrumenting enforcement actions. $S1$ and $S2$ are not inserted with any actions. Proof proceeds by induction on the size of the code block such that the lemma holds for all sub blocks of C with existing program points. As a result, we have $WP(C, post) \implies WP(C', post)$. The goal is to prove that after insertion of act between $S1$ and $S2$, $WP(C, post) \implies WP(C'', post)$ where $C'' \equiv C1'; act1; S1; act; S2; act2; C2'$. Based on the hypothesis and Lemma 45, $WP(act; S2; C2, post) \implies WP(act; S2; act2; C2', post)$. Based on the precondition on the local implication, $WP(S2; C2, post) \implies WP(act; S2; C2, post)$. As a result, $WP(S2; C2, post) \implies WP(act; S2; act2; C2', post)$. By applying the same strategy, we can prove that $WP(S1; S2; C2, post) \implies WP(act1; S1; act; S2; act2; C2', post)$. Due to Lemma 45, $WP(C1'; S1; S2; C2, post) \implies WP(C1'; act1; S1; act; S2; act2; C2', post)$. Moreover, $WP(C1, WP(S1; S2; C2, post)) \implies WP(C1', WP(S1; S2; C2, post))$ because of the inductive hypothesis. As result, $WP(C, post) \implies WP(C'', post)$. □

Theorem 48 *Assume programs F and F', F' is F with actions added at n program points* $s_1, ..., s_n$ *and that* $pre \implies WP(F, post)$. *If there exist some verification conditions* VC_1, VC'_1, ..., VC_n, VC'_n *respectively at point* $s_1, s_2, ..., s_n$ *in F and F' such that* $VC_k \implies VC'_k$ *for all k from 1 to* n, *then* $pre \implies WP(F', post)$.

Proof: From Lemma 47, $WP(F, post) \implies WP(F', post)$. Since $pre \implies WP(F, post)$, $pre \implies WP(F', post)$. □.

We present a method to construct the local implication for each instrumentation point below. With Theorem 48, correctness of the program is preserved as long as each obligation can be discharged. To encode the method into Frama-C, several technical challenges need to be overcome. Section 7.2.4 presents the overall process for generating the proof obligations. Section 7.2.5 points

out the name clashing problem and presents a method to unify names through data dependency graph.

7.2.4. Construction of the proof obligation

This section presents construction of the proof obligation. We assume the proof obligation of the original program has been constructed and discharged. Algorithm 5 is defined below. VC is a map from the program point to the verification condition of the original program at that point. $type$ is the type of the enforcement action, which can be either $suppression$, $insertion$ or $replacement$. If the action type is suppression, the code block from s to s' is removed where s' is the point after s in the program order; if the action type is insertion, the code block act is inserted at s; if the action type is replacement, the code block from s to s' is replaced by act. G is the CFG of the original program, which will be used to unify names as presented below.

First, the proof obligation at the point s in the original program is obtained by accessing the map VC. The overall obligation structure is represented as a tuple of σ and a set of sub-obligations denoted as vcs. Recall that σ is the map from variables to variable instances. Based on σ, we can obtain the CFG at s in the original program from G, denoted as $subG$. Then, the proof obligation for the modified program (denoted as (σ', vcs')) and the corresponding CFG (denoted as $subG'$) are obtained based on the type of action. If the action type is $suppression$, (σ', vcs') and $subG'$ are simply the verification condition and CFG at s'. If act is inserted at s, (σ', vcs') is computed using the WP calculus while $subG'$ at s can then be computed by taking into consideration act. The definitions of $updateCFG$ and $graphAt$ are omitted. The logic for replacement is similar except that the proof obligation is computed from s' instead of s. After obtaining the verification condition and CFG for the original and modified program, names of the variable instances are unified by the process presented below. Both vcs_update (for the original program) and vcs_update' (for the modified program) comply with $Sigma$.

Each sub-obligation of the original program is converted into a predicate by conjunction over its hypotheses and goal using the function $conj$. The local implication is constructed by adding this

predicate as a hypothesis in sub-obligations of the modified program. Note that the dot operator represents the list concatenation. The generated proof obligation can then be discharged. Note that more sophisticated analysis can be done to decide whether a verification condition in vcs_update may be helpful in proving the sub-obligation in vcs_update', which is out of the scope of this thesis. We also assume that preconditions of API calls are not added as sub-obligations because they are supposed to be enforced by the EM.

Algorithm 5 Construction of proof obligation

```
1:  procedure CONSTRUCTVC(F, s, s', type, act, VC, G)
2:      (σ, vcs) ← VC[s]
3:      subG ← graphAt(σ, G)
4:      if type == insertion then
5:          (σ', vcs') ← WP(act, σ, vcs)
6:          subG' ← updateCFG(act, subG)
7:      else
8:          if type == suppression then
9:              (σ', vcs') ← VC[s']
10:             subG' ← graphAt(σ', G)
11:         else
12:             if type == replacement then
13:                 (σ'', vcs'') ← VC[s']
14:                 subG'' ← graphAt(σ'', G)
15:                 (σ', vcs') ← WP(act, σ'', vcs'')
16:                 subG' ← updateCFG(act, subG'')
17:     (Sigma, vcs_update, vcs_update') ← nameUnify(σ, vcs, σ', vcs', subG, subG')
18:     for vc ∈ vcs_update do
19:         hypo ← conj(vc.hyps.[vc.goal])
20:         for vc2 ∈ vcs_update' do
21:             vc2.hyps ← vc2.hyps.hypo
        return (Sigma, vcs_update')
```

7.2.5. *Name unifying process*

Verification conditions from the original and the instrumented program are generated in different contexts. Restrictions on the implementation of enforcement actions guarantee that the stack and heap space are compatible between two versions of the program. However, the meaning of variable instances is not consistent. Two code snippets (denoted as P and P') are given in Figure 36 and Figure 37 to illustrate this problem. P' has an extra assignment to the variable *sum* at the label *L*

comparing to P. If we compute the verification condition for these two programs from the assertion (denoted as VC and VC'), two verification conditions will have different variable instances for the variable *sum*, as shown in Figure 38. However, when we combine VC and VC' together, the state of sum at L must be identical. As a result, we need to rename *sum_1* and *sum_2* into a common name. After renaming the independent variable instances, others are renamed subsequently following the data dependency. Generalizing this idea, we propose a name unifying process. The core idea is to generate a common set of variable instances that are independent from other variables and variable instances appearing in the obligations at the instrumentation point. All dependent variable instances can then be renamed by simply increasing the index according to the order of data dependency relations. We begin from the definition of data dependency graph over variable instances.

The dependency graph for the program to the instrumentation point s is constructed by analyzing each SSA when traversing the CFG from the post condition of the function to s. The left-hand-side (lhs) of the SSA depends on all variable instances appearing at the rhs of the SSA. We also need to consider the dependency relation specified in the auxiliary predicates generated for the function call and the `ITE` structure. For the function call $x = f(\mathit{args})$, x depends on the return value of function call. The post condition in the function contract may also define dependency relation between the return value and actual parameters. For the `ITE` structure, equality predicates are generated to specify the identical program state at the beginning of two branches. These predicates are also used to create the dependency graph. The dependency graph is guaranteed to be acyclic due to the properties of SSA construction. Figure 39 (a) and (b) respectively illustrate the dependency graph for P and P'.

To unify the name, we first find independent variable instances from two graphs and then rename them into a fresh name. Following this rule, *sum_1* and *sum_2* for variable *sum* in Figure 39 (a) and (b) are renamed into *sum_3*. Then, other variable instances can be renamed by traversing two graphs independently in a breadth-first manner. Variable *j* does not need to be renamed because its occurrences in two programs are consistent with respect to indices. The result of VC and VC' is illustrated in Figure 40. Note that we only use primary variables in the example. Although there

Figure 36: Original program *P*

```
 L:
    sum = sum + j*2;
  //@ assert even(sum);
  return 0;
}
```

Figure 37: *P'* with an extra assignment

```
 L:
    sum = sum + j*2;
    sum = sum + j*2;
  //@ assert even(sum);
  return 0;
}
```

Figure 38: Proof obligation VC and VC' before name unification

```
VC: (sum_0 = sum_1 + j_0*2) ->  (even sum_0)
VC': (sum_1 = sum_2 + j_0*2) /\ (sum_0 = sum_1 + j_0*2) ->  (even sum_0)
```

are no fundamental difficulties to extend the method to fully support aliases and heap structures, we will leave it as future work.

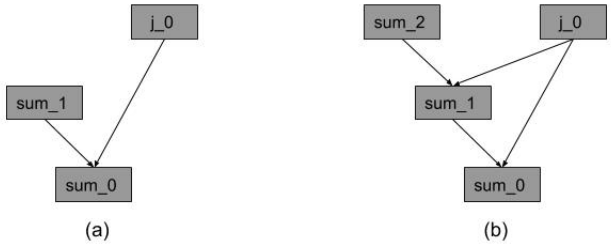

Figure 39: Dependency graphs for P and P'

7.2.6. Case study

We have implemented the method in Frama-C version 20.0. We use the variant of the *find* function illustrated in Figure 33 as the case study to demonstrate the application of our method.

The data structure of *Iterator* is defined in Figure 41. An iterator contains *iterator_c*, the array it points to. The field *iterator_pointer* denotes the next available index to access and *iterator_size* stores the size of the array. The specifications of the iterator API are defined in Figure 42. Defined as the predicate p_next, the specification of $next$ requires that the $iterator\_point$ is less than the size of the array to traverse, the return value is equal to the cell referred by the pointer and the

Figure 40: Proof obligation VC of P at L

```
VC: (sum_4 = sum_3 + j_0*2) ->  (even sum_4)
VC`: (sum_5 = sum_3 + j_0*2) /\ (sum_6 = sum_5 + j_0*2) ->  (even sum_6)
```

pointer is increased by 1. Similarly, the specification of $hasNext$ guarantees that the call returns 1 only when the pointer is smaller than the size of the array. The clause $assigns$ is used to restrict variables that can be modified by the function.

Figure 41: Data structure of iterator

```
typedef struct {
  int iterator_pointer;
  int iterator_size;
  collection iterator_c;
}Iterator;
```

To illustrate the application of our technique, we first prove the loop invariant in Figure 33 using Frama-C and then insert instructions to enforce the precondition of API specification of $next$ at call sites $L1$ and $L2$. By applying Theorem 48, we can build local implication and verify each of them independently. We assume the instrumented code block is an `ITE` structure that calls $call\_monitor\_next$ as shown in Figure 34 and Figure 35. While the `if` branch contains the actual enforcement actions to be verified, the code block in the `else` branch is identical to the one replaced by this `ITE` structure. As a result, we will ignore the `else` branch and generate verification conditions only for the `if` branch in the following discussion.

We first focus on $L1$ at which the value b is computed by calling $next$. For the enforcement action, we can either suppress the call to *next* or insert a call to *hasNext*. In both cases, the variable b needs to be assigned with a default value. However, because the proof target says if idx is not equal to -1, the value in the corresponding cell in the target array must be equal to v, the value of b actually does not influence the proof. As a result, neither suppression nor insertion will break the original proof. Nevertheless, if the instrumentation is incorrect, the proof will be broken. For instance, the instrumentation in Figure 34 mistakenly moves the action of updating the variable i in the `else` branch. When we try to prove the increment of i at each pass of the loop, the constructed local implication would take the form $old(i) + 1 > old(i) \implies i > old(i)$ where the

Figure 42: Predicates for *next* and *hasNext*

```
  predicate p_next(iterator *i, iterator *j, integer v) =
                        v == j->c[j->iterator_pointer]
                       && i ->iterator_pointer == j->iterator_pointer + 1;
*/
/*@
  predicate p_hasNext(iterator *i, integer v)=
      \result==1 ==> i->iterator_pointer < i.iterator_size
      && \result == 0 ==> i ->iterator_pointer >= i.iterator_size;
*/

 /*@
   requires i -> iterator_pointer < i.iterator_size;
   ensures p_next(i, \old(i), \result);
   assigns  i -> iterator_pointer;
 */
 int next(Iterator *i);

 /*@
    ensures p_hasNext(i, v);
    assigns \nothing;
 */
 int hasNext(Iterator *i);
```

pre- and post-condition are respectively the proof obligation at $L1$ for the original and the modified program. Since this obligation cannot be discharged, the user could discover that the instrumentation is problematic.

To demonstrate usability of constructing the local proof, we prove correctness of the implementation of insertions at $L2$ using Coq. For the call to *next* at $L2$, we insert a call to *hasNext* as the condition to check before calling $next$ and subsequent instructions. We denote the proof obligation at $L2$ for the original and the modified program as VC and VC', as shown in Figure 43 and 44, represented using the syntax of Coq. Since no instruction will be executed when *hasNext* returns false and instructions from $L1$ to $L2$ is orthogonal to the loop invariant to be proved, the precondition generated from the function call $hasNext$ and the verification condition are removed.

Accessing to the field of $iterator\_pointer$ of $iter$ is represented as the expression $iterator\_pointer\ iter$. $branch\ e1\ e2\ e3$ returns the value of $e2$ ($e3$) if $e1$ is evaluated to true (false). $collection\_array\ c$ represents the array pointed by c in which c is the field $iterator\_c$ of the iterator type. $next\_0$ and $next\_2$ represent the return value of calling $next$ in two versions of the code block. v_3 and v_4 indicate that the pointer value may be changed during the call of $next$. x_0 and x_1 are used

Figure 43: Verification condition VC of the original program at *L2*

```
let x_0 := iterator_pointer iter_12 in
(iter_10 = iter_11)
  /\ (value_3=next_0)
  /\ (x_0=(1+ iterator_pointer iter_11))
  /\ (p_next iter_12 iter_11 next_0)
  /\ (branch ((value_3=v%Z))
   (x_0=(1+idx_7)%Z) (idx_6=idx_7) % Z)
 /\ ({|
       iterator_c := iterator_c iter_11;
       iterator_size :=
         iterator_size iter_11;
       iterator_pointer := (v_3)%Z
       |}=iter_12)
 /\ ((idx_7 <> -1)->
 ((((collection_array (iterator_c
                        iter_12))))
   .[ (idx_7)%Z ] = v)
```

Figure 44: Verification condition VC' of the instrumented program at *L2*

```
  let x_1 := iterator_pointer iter_14 in
  (idx_9 <> -1) -> ((idx_6=idx_8)
  /\ (iter_10=iter_13)
  /\ (value_4=next_2)
  /\ (x_1= (1+ (iterator_pointer
                 iter_13)))
 /\ (p_next iter_14 iter_13 next_2)
 /\ ({|
       iterator_c := iterator_c iter_13;
       iterator_size := iterator_size
              iter_13;
       iterator_pointer := (v_4)%Z
       |}=iter_14)
 /\ (branch ((value_4=v%Z))
     (x_1=(1+idx_9)%Z)
     (idx_8=idx_9) % Z))
  -> ((((collection_array (iterator_c
            iter_14))))
  .[ (idx_9)%Z ] = v)
```

for succinct representation. Other identifiers taking the form a_b represent the instance of variable a with index b. Variable names have been unified and verification conditions in VC have been refactored as conjunctions as stated in Algorithm 5. Then VC is integrated as the precondition of VC'. To prove VC' in Coq using the information give in VC, we need to prove 1) $iter\_12$ is equivalent to $iter\_14$ with respect to the array they traverse and 2) idx_7 is identical to idx_9. Recall that the specification of $next$ defined in Figure 42 guarantees that only the field $iterator\_pointer$ will be changed, so 1) can be proved. 2) can also be proved trivially be rewriting terms. This is what we expect because the code at the branch where *hasNext* returns to true is identical to the original code. As a result, VC' can be proved in Coq by solely unfolding the definition and rewriting terms. On the other hand, building the proof obligation globally would make the user prove the same target from scratch or revising the existing proof, which requires duplicated work. It would be more problematic for complicated proof obligations.

7.3. Summary

In this chapter, we have extended the SMEDL framework to support specifying actions to be instrumented into the target program for reflexive adaptation. Exported events raised from the monitor are mapped to adaptation actions such as insertion of code block and suppression or replacement

of subsequent statements in the program. The semantic rules that consider the execution of adaptation actions were defined. Implementation and instrumentation of actions were proposed based on which a framework to verify the implementation of enforcement actions with respect to functional correctness of the target program was proposed. We presented a method to construct a proof obligation locally. If the generated obligation can be discharged, the enforcement action will not break the correctness proof of the original program. To connect verification conditions obtained from two programs, we proposed a name unifying algorithm.

CHAPTER 8 : Conclusion

In this thesis, we have presented a framework for efficient runtime verification and software adaptation. In this chapter, we summarize the contributions and discuss possible research directions.

8.1. Overview of work

SMEDL framework for runtime verification. In chapter 3, we proposed SMEDL, a formalism for runtime verification. A SMEDL specification is composed of a set of single monitor specifications connecting with each other following the specification defined in the architecture description. Single monitors use automata-based formalism for describing properties at multiple abstraction levels while a dynamically scalable monitor network is used to specify more sophisticated requirements. We implemented our framework in two ways. To monitor a critical system which treats correctness over performance, we utilized the Fiat framework to generate correct-by-construction monitor implementations by refinement from the mechanized semantics of single monitors. When efficiency is more important, C programs are generated as monitors. The evaluation in chapter 5 illustrates that it outperforms many existing RV tools in both online and offline settings.

Parametric monitoring using SMEDL. Parametric properties are widely used to describe behavior of large-scale systems. In chapter 4, we proposed a novel perspective of parametric monitoring using monitor network. By model transformation, we demonstrated that SMEDL can express the trace slicing algorithm in MOP. We then proposed a method to encode a quantified parameter list of QEA specifications as aggregation monitors. For a subset of QEA specifications, SMEDL monitors can generate identical verdicts more efficiently, which was demonstrated in chapter 5.

Monitoring time interval in asynchronous environments. Asynchronous deployment is necessary for monitoring distributed systems. However, under network delay and clock discrepancy, monitoring properties with time intervals is challenging. In chapter 6, we made initial efforts to parameterize the model of communication between a system and a monitor, which takes into consideration network delay, clock rate and clock skew. Based on this model, we proposed an interval

closure event and a method to decide which interval an incoming event belongs to. With this mechanism, users can separate concerns of monitoring intervals from the rest of property checking.

Reflexive adaptation framework. In chapter 7, we proposed an extension of SMEDL for reflexive software adaptation. Users can specify a map between events raised from a monitor and adaptation actions, including insertion, suppression or replacement of code blocks in the target program. We further defined the semantic rules to describe possible interference between the instrumented adaptation actions and the target program. To formally verify correctness of the adaptation implementation with respect to execution of the target program, we proposed a Hoare-logic based method to construct proof obligation locally, which has been implemented in the Frama-C framework.

8.2. Future directions

Asynchronous monitor network. In this thesis, we do not study behavior of asynchronous monitor network. Current implementations of asynchronous monitoring rely on underlying communication middleware, which do not have a formal semantics. Borrowing the idea from actor-based monitoring system [59] and generating Erlang programs [131] for asynchronous communication could be a promising direction.

Parallelization of SMEDL. In chapter 5, we have demonstrated time efficiency of SMEDL. However, there is still a lot of room for further optimization and one promising direction is parallelization within a synchronous set, which could be achieved at two levels. At the monitor level, multiple monitors handle an incoming event sequentially, which conforms to the semantics of trace slicing. However, two monitors can execute in parallel if they are not related through common parameters. At the instance level, parallel updates of instances of a monitor may also improve efficiency. It would be interesting to explore parallelization of SMEDL monitors and study how to guarantee compliance between the semantics and the implementation.

Overhead-aware deployment of SMEDL monitors. SMEDL supports hybrid deployment of monitors to achieve balance between overhead and performance. However, current setting requires users to decide the architecture manually. We have proposed an initial idea [140] to decide de-

ployment of a hierarchical monitor structure by analyzing the overhead brought by asynchronous communication and monitoring logic. It would be interesting to explore a general mechanism or process to achieve optimized deployment of monitors in an automatic or semi-automatic way.

Towards a more realistic method for monitoring time intervals. Our work in monitoring time intervals has two limitations. First, we can exactly determine when we have seen all the events only if the network delay is bounded. Second, we assume that we can precisely determine whether a given event is within the bounds of an interval or outside. In general, neither of these two assumptions are true. This means that the monitoring procedure needs to be augmented to accommodate the uncertainty. It would be interesting to explore a more realistic model for analyzing behaviors of monitors in asynchronous environments. For instance, if the network delay is unbounded, we can consider more widely used self-similar traffic models [93]. With this approach we can have a monitoring procedure with probabilistic guarantees of correctness. We will consider a three-valued semantics for the temporal logic, with the "unknown" value corresponding to an error, which may happen when we discover an event that belongs to an already-closed interval. Another issue comes from uncertainty in the system clock rate, which would lead to wrong decision on which interval the incoming event belongs to. In this case, we can also use a 3-valued semantics, with the third value representing the uncertainty whether the event occurs before or after the interval closure event. It remains to be seen whether the two three-valued approaches – the one capturing an error and the one capturing the ordering uncertainty – can be combined together in an effective checking procedure.

Software adaptation at multiple levels. Software adaptation involves both high-level repair process and low-level reflexive enforcement actions. In our work, we have integrated into SMEDL low-level enforcement actions. High-level repair may utilize more information such as sophisticated program profiling from multiple executions to synthesize repair solution. These two types of adaptation could have influence on each other. On one direction, execution of enforcement actions may give hints to the high-level repair. On the other direction, the modified code may require changes to monitors. It would be interesting to study how to combine them to achieve software adaptation with good performance.

Fully mechanized semantics and proofs. In this work, we have proposed the formal semantics of synchronous aspect of SMEDL and several theorems and lemmas. However, only the semantics of single monitors and related theorems in Section 3.1 have been encoded in Coq. It would be useful to mechanize the semantics of the whole SMEDL language and all proofs proposed in the thesis.

APPENDIX

A.1. Concrete syntax of SMEDL

A.1.1. Syntax of the single monitor specification

The EBNF (Extended Backus-Naur Form) of single monitors is given below. Some common definitions such as types and expressions are omitted. Not mentioned in the thesis, users can import C header files which provide side-effect-free *helper* functions. In the definition of a scenario, there is an optional field to state *final states*, which have been discussed in Chapter 5. In the definition of *event_declaration*, the comma is used as the delimiter of multiple *event_signature*.

```
⟨start⟩ ::= ⟨declaration⟩ [⟨helper_list⟩] [⟨state_section⟩] ⟨event_section⟩ ⟨scenario_section⟩
⟨declaration⟩ ::= 'object' ⟨identifier⟩ ';'
⟨helper_list⟩ ::= {⟨helper_definition⟩}*
⟨helper_definition⟩ ::= '#include' ⟨helper_filename⟩
⟨state_section⟩ ::= 'state:' {⟨state_declaration⟩}*
⟨state_declaration⟩ ::= ⟨identifier⟩ [ '=' ⟨signed_literal⟩ ] ';'
⟨event_section⟩ ::= 'events:' {⟨event_declaration⟩}*
⟨event_declaration⟩ ::= ('imported' | 'internal' | 'exported')
    ',' .{⟨event_signature⟩}+ ';'
⟨event_signature⟩ ::= ⟨identifier⟩ '(' ⟨type_list⟩ ')'
⟨scenario_section⟩ ::= 'scenarios:' {scenario}*
⟨scenario⟩ ::= ⟨identifier⟩ ':' ['finalstate' ⟨identifier⟩ ';'] {⟨transition⟩}*
⟨transition⟩ ::= ⟨identifier⟩ '->' ⟨step_definition_list⟩ [⟨else_definition⟩] ';'
⟨step_definition_list⟩ ::= ⟨step_definition⟩ '->' ⟨step_definition_list⟩
  | ⟨step_definition⟩ '->' ⟨identifier⟩
⟨step_definition⟩ ::= ⟨step_event_definition⟩ ['when' '(' expression ')'] [⟨action_list⟩]
⟨else_definition⟩ ::= 'else' [⟨action_list⟩] '->' ⟨identifier⟩
```

⟨*step_event_definition*⟩ ::= identifier '(' ⟨*identifier_list*⟩ ')' ;

⟨*action_list*⟩ ::= '{' ⟨*action_inner_list*⟩ '}'

⟨*action_inner_list*⟩ ::= ⟨*action*⟩ ';' ⟨*action_inner_list*⟩ | ⟨*action*⟩ | ()

⟨*action*⟩ ::= ⟨*assignment*⟩ | ⟨*raise_stmt*⟩

⟨*assignment*⟩ ::= ⟨*identifier*⟩ '=' ⟨*expression*⟩

⟨*raise_stmt*⟩ ::= 'raise' ⟨*identifier*⟩ '(' ⟨*expression_list*⟩ ')' ;

⟨*expression_list*⟩ ::= ',' .{⟨*expression*⟩}*

A.1.2. Syntax of the architecture description

The architecture description imports files of all single monitor specifications for type checking. Then, the interface of all monitors are declared. Note that identities of a monitor are defined as a list of types without names, which means they cannot be used in monitoring logic. In the definition of *event connection specification*, wild cards are represented by '*'. An identity parameter of a monitor and an attribute of an event are respectively represented by $\#.n$ and $\$.n$ where *n* is the position in the parameter or attribute list. Synchronous set is specified in $syncset\_decl$.

⟨*start*⟩ ::= ⟨*declaration*⟩ ';' {(⟨*import_stmt*⟩ | ⟨*monitor_decl*⟩ | ⟨*event_decl*⟩ | ⟨*syncset_decl*⟩
| ⟨*connection_defn*⟩)';'}*

⟨*declaration*⟩ ::= 'system' ⟨*identifier*⟩

⟨*import_stmt*⟩ ::= 'import' ⟨*smedl_filename*⟩

⟨*monitor_decl*⟩ ::= 'monitor' ⟨*identifier*⟩ '(' ⟨*type_list*⟩ ')' ['as' ⟨*identifier*⟩]

⟨*event_decl*⟩ ::= 'event' ⟨*identifier*⟩ '(' ⟨*type_list*⟩ ')'

⟨*syncset_decl*⟩ ::= 'syncset' ⟨*identifier*⟩ '{' ⟨*identifier_list*⟩ '}'

⟨*connection_defn*⟩ ::= [⟨*identifier*⟩ ':'] ⟨*source_spec*⟩ '=>' ⟨*target_spec*⟩

⟨*source_spec*⟩ ::= [⟨*identifier*⟩ '.'] ⟨*identifier*⟩

⟨*target_spec*⟩ ::= ⟨*target_event*⟩

⟨*target_event*⟩ ::= ⟨*identifier*⟩ [‘`[`’ ⟨*wildcard_parameter_list*⟩ ‘`]`’] ‘`.`’ ⟨*identifier*⟩ [‘`(`’ ⟨*parameter_list*⟩ ‘`)`’]

⟨*wildcard_parameter_list*⟩ ::= ‘`,`’.{⟨*wildcard_parameter*⟩}*

⟨*parameter_list*⟩ ::= ‘`,`’.{⟨*parameter*⟩}*

⟨*wildcard_parameter*⟩ ::= ⟨*parameter*⟩ | ‘`*`’

⟨*parameter*⟩ ::= ‘`#.`’ ⟨*natural*⟩ | ‘`$.`’ ⟨*natural*⟩

⟨*natural*⟩ ::= /[0-9]+/

A.2. SMEDL examples

This section lists SMEDL specifications appearing in Chapter 5. *Watertank* and *TrackQuality* are not included. Each specification contains a set of single monitor specifications and an architecture description when the specification contains multiple monitors. For simplicity, *import_stmt* clause in the architecture descriptions are omitted.

A.2.1. BasicCar

The policy of a car is modeled as a SMEDL specification *Car* below. When the policy is violated, an exported event *violation* is raised.

```
object Car;
events:
  imported toggle_lights();
  imported toggle_wipers();
  imported accelerate();
  imported create();
  imported destroy();
  exported violation();
scenarios:
  sce1:
  init -> create() -> start;
  init -> toggle_lights() {raise violation();} -> init;
  init -> toggle_wipers() {raise violation();} -> init;
  init -> accelerate() {raise violation();} -> init;
  init -> destroy() {raise violation();} -> init;
  start -> toggle_lights() -> start;
  start -> toggle_wipers() -> start;
  start -> accelerate() -> start;
  start -> create() {raise violation();} -> start;
  start -> destroy() -> done;
  done -> create() -> start;
  done -> toggle_lights() {raise violation();} -> done;
  done -> toggle_wipers() {raise violation();} -> done;
  done -> accelerate() {raise violation();} -> done;
  done -> destroy() {raise violation();} -> done;
```

A.2.2. UnsafeFile

The monitor *UnsafeFile* checks whether file operations are legal. The *end* event is used to test whether the file has been closed when the program exits.

```
object UnsafeFile;
events:
  imported open();
  imported close();
  imported put();
  imported end();
  exported violation();
scenarios:
  sce1:
  init -> open() -> start;
  init -> close() {raise violation();} -> init;
  init -> put() {raise violation();} -> init;
  start -> put() -> start;
  start -> close() -> done;
  start -> open() {raise violation();} -> start;
  done -> put() {raise violation();} -> done;
  start -> end() {raise violation();} -> start;
```

A.2.3. GrantCancel

The monitor *GrantCancelResource* detects a violation of the *GrantCancel* property when a resource is granted multiple times or cancelled by a task not owning it. The monitor keeps track of granting and canceling of a resource to a task. As a result, the number of instances corresponds to the number of resources during runtime.

```
object GrantCancelResource;
state:
    int task;
events:
    imported grant(int);
    imported cancel(int);
    exported violation();
scenarios:
    main:
        free -> grant(granted_task) {task = granted_task;} -> granted;
        granted -> grant(granted_task) {raise violation();} -> granted;
        granted -> cancel(granted_task) when (granted_task == task) -> free
            else {raise violation();} -> granted;
```

A.2.4. NestedCommand

The specification has two monitors. *FirstCommand* is created whenever a new command is issued. A command issued later will be combined with all existing commands by creating an instance of *CommandPair*. If an previously issued command *succeeds* before an later one, a *violation* is detected.

```
object FirstCommand;
events:
    imported command(int);
    imported success();
    exported second_command(int);
scenarios:
    export_pairs:
        finalstate done;
        running -> command(id) {raise second_command(id);} -> running;
        running -> success() -> done;
```

```
object CommandPair;
events:
    imported first_success();
    imported second_success();
    exported violation();
scenarios:
    main:
        finalstate done;
        both_running -> second_success() -> done;
        both_running -> first_success() {raise violation();} -> done;
```

```
system NestedCommands;

monitor FirstCommand(int);
monitor CommandPair(int, int);

cmd1: command => FirstCommand[*].command($0);
cmd1: command => FirstCommand($0);
cmd2: FirstCommand.second_command => CommandPair(#0, $0);
succ: succeed => CommandPair[*, $0].second_success();
succ: succeed => CommandPair[$0, *].first_success();
succ: succeed => FirstCommand[$0].success();
```

A.2.5. ResourceLifeCycle

The monitor *Resource* raises a *violation* event if three stages of a resource are not issued in order. The *end* event is used to check whether a resource is eventually cancelled when the program exits.

```
object Resource;
state:
    int task;
events:
    imported request();
    imported deny();
    imported grant();
    imported cancel();
    imported rescind();
    imported end();
    exported violation();
scenarios:
    main:
        free -> request() -> requested;
        free -> deny() {raise violation();} -> fail;
        free -> grant() {raise violation();} -> fail;
        free -> rescind() {raise violation();} -> fail;
        free -> cancel() {raise violation();} -> fail;
        free -> end() -> pass;
        requested -> deny() -> free;
        requested -> grant() -> granted;
        requested -> request() {raise violation();} -> fail;
        requested -> rescind() {raise violation();} -> fail;
        requested -> cancel() {raise violation();} -> fail;
        requested -> end() -> pass;
        granted -> cancel() -> free;
        granted -> rescind() -> granted;
        granted -> request() {raise violation();} -> fail;
        granted -> deny() {raise violation();} -> fail;
        granted -> grant() {raise violation();} -> fail;
        granted -> end() {raise violation();} -> fail;
```

A.2.6. RespectConflict

To monitor behavior of two conflicting entities (denoted as *A* and *B*) with respect to the resource, we need to use two monitors *RespectA* and *RespectB*. Both of them are *Respect* but with reversed identities, as shown in the architecture description *RespectArch*. If a resource has been granted to either A and B, it cannot be granted to the other one. Moreover, a resource can only be cancelled by

the entity to which it has been granted to.

```
object Respect;
state:
   int state;
   int pre;
   int post;
events:
   imported conflict(int, int);
   imported grant(int);
   imported cancel(int);
   exported violation();
scenarios:
    main:
         s1 -> conflict(pr, po) {pre = pr; post = po; state = -1;} -> s1;
         s1 -> grant(v) when (state == -1 && v == pre) {state = pre;} -> s1;
         s1 -> grant(v) when (state == -1 && v == post) {state = post;} -> s1;
         s1 -> grant(v) when (state != -1) {raise violation();} -> s1;
         s1 -> cancel(v) when (state != v) {raise violation();} -> s1;
         s1 -> cancel(v) when (state == v) {state = -1;} -> s1;
```

```
system RespectArch;

monitor RespectA(int, int) as Respect;
monitor RespectB(int, int) as Respect;

con: conflict => RespectA[$0, $1].conflict($0, $1);
con: conflict => RespectB[$1, $0].conflict($1, $0);
gr:  grant => RespectA[$0, *].grant($0);
gr:  grant => RespectB[*, $0].grant($0);
cl:  cancel => RespectA[$0, *].cancel($0);
cl:  cancel => RespectB[*, $0].cancel($0);
```

A.2.7. Auction

```
object Auctionmonitor;
state:
  float reserve_price;
  float current_price = 0;
  float duration;
  float days_passed = 0;
events:
  imported create_auction(int, int, int);
  imported bid(int, int);
  imported sold(int);
  imported end_of_day();
  exported alarm_recreation();
  exported alarm_low_bid();
  exported alarm_sold_early();
  exported alarm_not_sold();
  exported alarm_action_after_end();
  exported alarm_action_before_start();
scenarios:
main:
  init -> create_auction(item, minimum, period) {reserve_price = minimum; duration = period;} -> bidding;
  bidding -> bid(item, amount) {current_price = amount;} -> above_reserve;
  bidding -> sold(item) {raise alarm_sold_early();} -> error;
  bidding -> end_of_day() when (days_passed < duration - 1) {days_passed++;} -> bidding
          else -> done;
  bidding -> create_auction(item, minimum, period) {raise alarm_recreation();} -> error;
  above_reserve -> sold(item) when (current_price < reserve_price ) {raise alarm_low_bid();} -> error
            else -> done;
  above_reserve -> bid(item, amount) when (amount > current_price) {current_price = amount;} -> above_reserve;
  above_reserve -> end_of_day() when (days_passed < duration - 1) {days_passed++;} -> above_reserve
          else -> done;
  above_reserve -> create_auction(item, minimum, period) {raise alarm_recreation();} -> error;
  done -> end_of_day() -> done;
  done -> bid(item, amount) {raise alarm_action_after_end();} -> error;
  done -> sold(item) {raise alarm_action_after_end();} -> error;
  done  -> create_auction(item, minimum, period) {raise alarm_recreation();} -> error;
```

A.2.8. CandidateSelection

The specification principally follows Figure 8 (b) and Figure 9 with slight modifications such as removal of *mon0* and combination of mvp and $aMon_c$.

```
object Mvp;
state:
  int canNum = 0;
events:
  imported member(string, string);//(voter, party)
  imported candidate(string, string);//(candidate, party)
  imported countcan();//(voter, party)
  imported valid();//(voter, party)
  imported end();
  internal check();
  exported createVCP(string);//(voter, candidate, party)
  exported result(int);
  exported addP();
scenarios:
  sce:
    init -> member(v, p) {raise addP();} -> start;
    start -> member(v, p) -> start;
    start -> candidate(c, p) {raise createVCP(c); canNum = canNum + 1; } -> start;
  sce1:
    start -> valid() {canNum = canNum - 1;} -> start;
    start -> end() {raise check();} -> start;
  sce2:
    start -> check() when (canNum == 0) {raise result(1); canNum = 0;} -> start;
    start -> check() when (canNum > 0) {raise result(0); canNum = 0;} -> start;
```

```
object Mvcp;
events:
  imported createVCP();//(voter, candidate, party)
  imported rank(string, string, int);//(voter, candidate, rank)
  exported valid();//(voter, party)
scenarios:
  sce:
    init -> createVCP() -> start;
    start -> rank(v, c, i) {raise valid();} -> end;
```

```
object AMonP;
state:
  int pNum = 0;
  int res = 0;
events:
  imported addP();
  imported inRes(int);
  internal check();
  exported result(int);//
  exported addV();
scenarios:
  sce:
    init -> addP() {pNum = pNum + 1; raise addV(); } -> start;
    start -> addP() {pNum = pNum + 1; } -> start;
  sce1:
    start -> inRes(i) when (pNum > 1) {res = res + i; pNum = pNum - 1; } -> start;
    start -> inRes(i) when (pNum == 1) {res = res + i; raise check(); pNum = pNum - 1;
    } -> start;
  sce2 :
    start -> check() when (res > 0) {raise result(1);} -> start;
    start -> check() when (res == 0) {raise result(0);} -> start;
```

```
object AMonV;
state:
  int vNum = 0;
  int vNumTemp = 0;
  int res = 0;
events:
  imported addV();
  imported inRes(int);
  exported result(int);
scenarios:
  sce:
    start -> addV() {vNum ++; } -> start;
  sce1:
    init -> inRes(i) when (vNum > 1) {res = res + i; vNumTemp = vNum; vNum --; } -> start;
    init -> inRes(i) when (vNum == 1) {res = i - vNum; raise result(res); vNum = 0;}
    -> start;
    start -> inRes(i) when (vNum > 1) {res = res + i; vNum --; } -> start;
    start -> inRes(i) when (vNum == 1) { res = res + i - vNumTemp; raise result(res);
    vNum = 0;} -> init;
```

```
system CanSys;

monitor Mvp(string, string);
monitor Mvcp(string, string, string);
monitor AMonP(string);
monitor AMonV();

ch1: member => Mvp[$1, $0].member($0, $1);
ch2: candidate => Mvp[$1,*].candidate($0,$1);
ch3: end => Mvp[*,*].end;
ch5: Mvcp.valid => Mvp[#2,#1].valid;
ch6: Mvp.createVCP => Mvcp[$0, #1, #0].createVCP;
ch7: rank => Mvcp[$1, $0, *].rank($0, $1, $2);
ch8: Mvp.addP => AMonP[#1].addP;
ch9: Mvp.result => AMonP[#1].inRes($0);
ch10: AMonP.addV => AMonV.addV;
ch11: AMonP.result => AMonV.inRes($0);
```

A.2.9. SqlSanitizer

An instance of *Sql_input* is created for an input not derived from any other ones. For a derived input, an instance of *Sql_derived* is created, which is parameterized by the identity of the original and the derived input. An input can also be derived from another derived input, which is modeled by the last transition in *Sql_derived*. The operation of sanitization can be propagated from an input to its descendants.

```
object Sql_input;
state:
     int is_san = 0;
events:
  imported input(int);
  imported sanitize();
  imported use();
  imported derive(int);
  exported derivation_out(int);
  exported propagate_sanitization();
  exported violation();
scenarios:
main:
     s1 -> input(s) -> s2;
     s2 -> use() when (is_san == 0) {raise violation();} -> s2;
     s2 -> sanitize() {is_san = 1; raise propagate_sanitization();} -> s2;
     s2 -> derive(t) {raise derivation_out(t);} -> s2;
```

```
object Sql_derived;
state:
     int is_san = 0;
events:
  imported sanitize();
  imported use();
  imported derive(int);
  imported derivation_in(int, int);
  exported derivation_out(int);
  exported propagate_sanitization();
  exported violation();
scenarios:
main:
     s1 -> derivation_in(t, s) -> s2;
     s2 -> use() when (is_san == 0) {raise violation();} -> s2;
     s2 -> sanitize() {is_san = 1; raise propagate_sanitization();} -> s2;
     s2 -> derive(t) {raise derivation_out(t);} -> s2;
```

```
system Sql_sanitize;

monitor Sql_input(int);
monitor Sql_derived(int,int);

ch1: input => Sql_input[$0].input($0);
ch2: derive => Sql_input[$0].derive($1);
ch2: derive => Sql_derived[$0, *].derive($1);
ch3: use => Sql_input[$0].use;
ch3: use => Sql_derived[$0, *].use;
ch4: sanitise => Sql_input[$0].sanitize;
ch4: sanitise => Sql_derived[$0, *].sanitize;
ch5: Sql_input.derivation_out => Sql_derived[$0, #0].derivation_in($0, #0);
ch7: Sql_input.propagate_sanitization => Sql_derived[*, #0].sanitize;
ch8: Sql_derived.propagate_sanitization => Sql_derived[*, #0].sanitize;
ch9: Sql_derived.derivation_out => Sql_derived[$0, #0].derivation_in($0, #0);
```

A.2.10. Banking-1

The MFOTL formula of this property is *ALWAYS FORALL c, t, a. trans(c,t,a) AND 2000 < a IMPLIES EVENTUALLY[0,6) report(t)*. MonPoly explicitly handles time and all events are attached

with a timestamp, which makes it efficient to check properties with time intervals. In contrast, SMEDL monitors do not have an internal clock to count ticks, which will lead to incorrect result or delay in outputting verdicts. To handle this issue, we insert a *tick* event between two consecutive events with increased timestamp. Whenever *TransactionReport* receives a tick event, it will check the time difference between the current timestamp and the timestamp of the last transaction (stored in the state variable *t0*).

```
object TransactionReport;
state:
    int t0;
events:
    imported trans(int, int, int, int);
    imported report(int, int);
    imported tick(int);
    exported violation();
scenarios:
    main:
        s1 -> trans(ts, c, t, a) when (a > 2000) {t0 = ts;} -> s2;
        s1 -> report(ts, t) -> s1;
        s2 -> report(ts, t) when (ts - t0 > 5) {raise violation();} -> s1
              else -> s1;
        s2 -> tick(ts) when(ts - t0 > 5) {raise violation();} -> s1;
```

A.2.11. Banking-2

The MFOTL formula of this property is *ALWAYS FORALL c, t, a. trans(c,t,a) AND 2000 < a IMPLIES ONCE[2,21) EXISTS e. auth(e,t)*. The tricky part is that multiple *auth* events may arrive and as long as one of them satisfies the timing requirement, the property is not violated. However, SMEDL monitors do not directly support arrays or dynamic data structure. To solve this issue, we use an integer typed *window* to represent a list of consecutive timestamps. A bit is set when an *auth* happens at that timestamp. The least (most) significant bit represents the left (right) boundary of the window and *t0* and *t1* are the corresponding time stamps. A *violation* is raised when a *trans* event arrives: 1) before any *auth* events or 2) less than 2 days after the earliest *auth* event or greater than 20 days after the latest *auth* event. When an *auth* event arrives, the right boundary of the

window is updated and the corresponding bit is set with its timestamp. For the left boundary, 1) if the timestamp is 20 days greater than the left boundary but not 20 days greater than the right boundary, the left boundary moves right to its nearest time that corresponds to an *auth* event; 2) if the timestamp is 20 days greater than the right boundary, the window size shrinks to 1. Note that there is a corner case in which the transaction happens 1 day after the nearest authorization and the next earliest authorization happens more than 20 days ago. In that case, a *violation* event is also raised.

```
object TransAuth;
#include "helper.h"
state:
    int t0;
    int t1;
    int window;
events:
    imported trans(int, int, int);
    imported auth(int, int);
    internal check(int, int, int);
    exported violation();
scenarios:
    main:
        s1 -> auth(ts, t) {t0 = ts; t1 = ts; window = 1;} -> s2;
        s1 -> trans(ts,  t, a) when (a > 2000) {raise violation();} -> s1;
        s2 -> trans(ts,  t, a) when (a > 2000 && (ts - t0 < 2 ||
ts - t1 > 20)) {raise violation();} -> s2
              else {raise check(ts, t0 + log2((window - 1) & -(window - 1), a);}-> s2;
        s2 -> auth(ts, t) when (ts - t0 <= 20) {t1 = ts;
        window = window | (1 << (ts - t0));} -> s2;
        s2 -> auth(ts, t) when (ts - t0 > 20 && ts - t1 <= 20) {t0 = t0 + log2((window - 1)
        & -(window - 1)); t1 = ts;
        window = window | (1 << (ts - t0));} -> s2;
        s2 -> auth(ts,t) when (ts - t1 > 20) {t0 = ts; t1 = ts; window = 1;} -> s2;
    corner:
        s1 -> check(ts, n, a) when (a > 2000 && ts - t1 == 1 && ts - t0 > 20 && n == t1)
        {raise violation();} -> s1
```

A.2.12. Publish

The MFOTL formula of this property is *ALWAYS FORALL a, f. publish(a,f) IMPLIES (NOT acc_F(a) SINCE[0,*) acc_S(a)) AND ONCE[0,10] EXISTS m. (NOT mgr_F(m,a) SINCE[0,*) mgr_S(m,a)) AND approve(m,f).* *acc_S(A)* and *acc_F(A)* indicate starting and ending of *A* being an accountant; *mgr_S(M,A)* and *mgr_F(M,A)* indicate starting and ending of *M* being a manager of *A*. The property says that if *a* publishes a report *f*, then *a* must be an accountant and there must exist an approval from *m* within 10 days before publishing and *m* must be the manager of *a* when the approval happens. The property is violated in the following cases: 1) a report *f* is published by a person *a* who is not an accountant; 2) there is no approval of *f* within the specified range; 3) there is no approval of *f* from a person *m* who is the manager of *a* who publishes *f*.

Parameterized by a person *a* (not used in the monitoring logic), *Account* keeps track of the status of *a*. When *state* is 1, *a* is not an accountant. The state is sent to *Validation* through *is_acc*, which is required when *a* publishes a report.

```
object Accountant;
state:
    int state;
events:
    imported acc_S();
    imported acc_F();
    imported checkAcc(int, int);
    exported is_acc(int, int, int);
scenarios:
    main:
        s1 -> acc_S() {state = 0;} -> s1;
        s1 -> acc_F() {state = 1;} -> s1;
        s1 -> checkAcc(m, f) {raise is_acc(m, f, state);} -> s1;
```

Similar to *Account*, *Manager(m)* is parameterized by a pair of (*m*, *a*): *m* is the manager of *a* iff *state* is 0. Whenever *m* approves a report *f*, we send the status of (m, a) to *Validation*.

```
object Manager;
state:
    int state;
events:
    imported mgr_S();
    imported mgr_F();
    imported internal_approve(int);
    exported createValidation(int, int);
scenarios:
    main:
        s1 -> mgr_S() {state = 0; } -> s1;
        s1 -> mgr_F() {state = 1; } -> s1;
        s1 -> internal_approve(f) {raise createValidation(f, state);} -> s1;
```

Parameterized by a report *f*, *Approve* does three things: 1) raise *final_result* to indicate violation of the property when a report *f* is not approved before publishing; 2) creation of an instance of *Aggregation(f)* and 3) trigger execution of *Manager* and *Internal_Approve* when *f* is approved by a manager.

```
object Approve;
events:
    imported approve(int, int, int);
    imported publish(int, int, int);
    exported internal_approve(int, int, int);
    exported create_aggregator();
    exported final_result(int);
scenarios:
    main:
        s0 -> approve(ts, m, f) {raise create_aggregator();
        raise internal_approve(ts, m, f); } -> s1;
        s0 -> publish(ts, a, f) {raise final_result(1);} -> s1;
        s1 -> approve(ts, m, f) {raise internal_approve(ts, m, f);} -> s1;
```

Internal_Approve is parameterized by (m, f). When *m* approves *f*, an *incr* event is sent to *Aggregation* to count the number of managers that have approved *f*. The state variable *t0* stores the time of approval. When *f* is approved, it raises an event $internal\_publish(a, v)$ in which *a* is the accountant that publishes *f* and *v* is 0 (1) when the nearest approval (does not) happen within 10 days

before publishing. *check_acc* is sent to *Accountant* to check whether *a* is an accountant at the time of publishing, as shown above.

```
object Internal_Approve;
state:
    int t0;
events:
    imported internal_approve(int);
    imported publish(int, int, int);
    exported internal_publish(int, int);
    exported check_acc(int);
    exported check_mgr(int);
    exported incr();
scenarios:
    main:
        s1 -> internal_approve(ts) {t0 = ts; raise incr();} -> s1;
        s1 -> publish(ts, a, f) when (ts < t0 || ts - t0 > 10)
        {raise internal_publish(a, 1); } -> s1
              else {raise internal_publish(a, 0); raise check_acc(a);} -> s1;
```

Validation is parameterized by (m, a, f). Variable *state* is 0 iff *m* is the manager of *a*. The attribute of *internal_publish* and *acc_result* are explained above. If the attribute of *violation* is 0, the triple satisfies the relation specified in the property.

```
object Validation;
state:
    int state;
events:
    imported internal_publish(int);
    imported createValidation(int);
    imported acc_result(int);
    exported violation(int);
scenarios:
    main:
        s1 -> createValidation(st) {state = st;} -> s1;
        s1 -> internal_publish(v) when (state == 1 || v == 1) {raise violation(1);} -> s1
              else -> s3;
        s3 -> acc_result(v) {raise violation(v);} -> s1;
```

Aggregation collects results from *Validation*. For each report *f*, we only need to find one pair of (*m*, *a*) that satisfies of the property. Note that we do not consider how to reset the state of monitors after a verdict for a report has been output.

```
object Aggregation;
state:
    int num = 0;
    int num0;
events:
    imported create_aggregator();
    imported violation(int);
    imported incr();
    exported final_result(int);
scenarios:
    main:
        s0 -> create_aggregator() -> s1;
        s1 -> incr() {num = num + 1;} -> s1;
        s1 -> violation(v) when (v == 0) -> s3;
        s1 -> violation(v) when (v == 1 && num > 1) {num0 = num - 1;} -> s2;
        s1 -> violation(v) when (v == 1 && num == 1) {raise final_result(1);} -> s1;
        s2 -> violation(v) when (v == 1 && num0 > 1) {num0 = num0 - 1;} -> s2;
        s2 -> violation(v) when (v == 1 && num0 == 1) {raise final_result(1);} -> s1;
        s2 -> violation(v) when (v == 0) -> s3;
```

```
system Publish;

monitor Accountant(int);
monitor Manager(int, int);
monitor Approve(int);
monitor Internal_Approve(int, int);
monitor Validation(int, int, int);
monitor Aggregation(int);

syncset Pub {Accountant, Manager, Approve, Internal_Approve, Validation, Aggregation};

ch1: acc_S => Accountant[$1].acc_S;
ch2: acc_F => Accountant[$1].acc_F;
ch3: mgr_S => Manager[$1, $2].mgr_S;
ch4: mgr_F => Manager[$1, $2].mgr_F;

ch5: approve => Approve[$2].approve($0, $1, $2);
ch6: publish => Approve[$2].publish($0, $1, $2);
ch6: publish => Internal_Approve[*, $2].publish($0, $1, $2);

ch7: Accountant.is_acc => Validation[$0,$1,#0].acc_result($2);

ch9: Approve.internal_approve => Internal_Approve[$1,#0].internal_approve($0);
ch9: Approve.internal_approve => Manager[$1,*].internal_approve($2);
ch10: Approve.create_aggregator => Aggregation[#0].create_aggregator;

ch11: Internal_Approve.internal_publish => Validation[#0, #1, $0].internal_publish($1);
ch12: Internal_Approve.check_acc => Accountant[$0].checkAcc(#0, #1);
ch14: Internal_Approve.incr => Aggregation[#1].incr;
ch15: Validation.violation => Aggregation[#1].violation($0);
ch17: Manager.createValidation => Validation[#0,$0,#1].createValidation($1);
```

A.2.13. UnsafeMapIter

The specification follows Figure 7 with slight modifications: 1) *mon0* is removed and 2) *useI* is not dispatched to *MC*.

```
object MC;
events:
  imported createC();//mid, cid
  imported createI(pointer);//cid, iid
  exported createMCI(pointer);
scenarios:
sce1:
  init -> createC() -> start;
  start ->createI(i) {raise createMCI(i);} -> start;
```

```
object MCI;
events:
  imported createMCI();//iid
  imported updateM();//mid
  imported useI();//iid
  exported violation();
scenarios:
sce1:
  init -> createMCI() -> start;
  start -> useI() -> start;
  start -> updateM() -> updateM;
  updateM -> useI() {raise violation();} -> error;
```

```
system MapArch;

monitor MC(pointer, pointer);
monitor MCI(pointer, pointer, pointer);

syncset sync {MC, MCI};

ch1: createC => MC[$0,$1].new_mc;
ch2: createI => MC[*,$0].createI($1);
ch3: MC.createMCI => MCI[#0,#1,$0].createMCI;
ch4: updateM=> MCI[$0,*,*].updateM;
ch5: useI=> MCI[*,*,$0].useI;
```

BIBLIOGRAPHY

[1] Apache storm. `https://storm.apache.org/`. Accessed: 2021-09-23.

[2] Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1985.

[3] Irem Aktug and Katsiaryna Naliuka. ConSpec—a formal language for policy specification. *Science of Computer Programming*, 74(1-2):2–12, 2008.

[4] Irem Aktug, Mads Dam, and Dilian Gurov. Provably correct runtime monitoring. In *International Symposium on Formal Methods*, pages 262–277. Springer, 2008.

[5] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege De Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *ACM SIGPLAN Notices*, volume 40, pages 345–364. ACM, 2005.

[6] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In *European Symposium on Programming*, pages 15–40. Springer, 2016.

[7] Shaun Azzopardi, Christian Colombo, Jean-Paul Ebejer, Edward Mallia, and Gordon J Pace. Runtime verification using Valour. *RV-CuBES*, 2017.

[8] Roger S Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. *arXiv preprint cs/0612115*, 2006.

[9] Howard Barringer and Klaus Havelund. Internal versus external DSLs for trace analysis. In *International Conference on Runtime Verification*, pages 1–3. Springer, 2011.

[10] Howard Barringer and Klaus Havelund. TraceContract: A Scala DSL for trace analysis. In *International Symposium on Formal Methods*, pages 57–72. Springer, 2011.

[11] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 44–57. Springer, 2004.

[12] Howard Barringer, Alex Groce, Klaus Havelund, and Margaret Smith. Formal analysis of log files. *Journal of aerospace computing, information, and communication*, 7(11):365–390, 2010.

[13] Howard Barringer, David Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *Journal of Logic and Computation*, 20(3):675–706, 2010.

[14] Howard Barringer, Ylies Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *International Symposium on Formal Methods*, pages 68–84. Springer, 2012.

[15] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In *Lectures on Runtime Verification*, pages 1–33. Springer, 2018.

[16] David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu. Monitoring of temporal first-order properties with aggregations. In *International Conference on Runtime Verification*, pages 40–58. Springer, 2013.

[17] David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu. On real-time monitoring with imprecise timestamps. In *International Conference on Runtime Verification*, pages 193–198. Springer, 2014.

[18] David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. Monitoring metric first-order temporal properties. *Journal of the ACM (JACM)*, 62(2):15, 2015.

[19] David A Basin, Felix Klaedtke, and Eugen Zalinescu. The MonPoly monitoring tool. *RV-CuBES*, 3:19–28, 2017.

[20] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI C specification language, 2008.

[21] Andreas Bauer and Ylies Falcone. Decentralised LTL monitoring. In *International Symposium on Formal Methods*, pages 85–100. Springer, 2012.

[22] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14, 2011.

[23] Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. From propositional to first-order monitoring. In *International Conference on Runtime Verification*, pages 59–75. Springer, 2013.

[24] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with Polymer. In *ACM SIGPLAN Notices*, volume 40, pages 305–314. ACM, 2005.

[25] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[26] Nataliia Bielova and Fabio Massacci. Predictability of enforcement. In *International Symposium on Engineering Secure Software and Systems*, pages 73–86. Springer, 2011.

[27] Nataliia Bielova and Fabio Massacci. Do you really mean what you actually enforced? *International Journal of Information Security*, 10(4):239–254, 2011.

[28] Nataliia Bielova and Fabio Massacci. Iterative enforcement by suppression: Towards practical enforcement theories. *Journal of Computer Security*, 20(1):51–79, 2012.

[29] Jan Olaf Blech, Ylies Falcone, and Klaus Becker. Towards certified runtime verification. In *International Conference on Formal Engineering Methods*, pages 494–509. Springer, 2012.

[30] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot–a technique for cheap recovery. *arXiv preprint cs/0406005*, 2004.

[31] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[32] Ian Cassar and Adrian Francalanza. On synchronous and asynchronous monitor instrumentation for actor-based systems. *arXiv preprint arXiv:1502.03514*, 2015.

[33] Pedro Celis, Per-Ake Larson, and J Ian Munro. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 281–288. IEEE, 1985.

[34] Feng Chen and Grigore Roşu. Parametric trace slicing and monitoring. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 246–261. Springer, 2009.

[35] Zhe Chen, Zhemin Wang, Yunlong Zhu, Hongwei Xi, and Zhibin Yang. Parametric runtime verification of C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 299–315. Springer, 2016.

[36] Adam Chlipala, Benjamin Delaware, Samuel Duchovni, Jason Gross, Clément Pit-Claudel, Sorawit Suriyakarn, Peng Wang, and Katherine Ye. The end of history? using a proof assistant to replace language design with library design. In *SNAPL'17: 2nd Summit on Advances in Programming Languages*, 2017.

[37] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 88–98, 2001.

[38] Christian Colombo and Yliès Falcone. Organising LTL monitors over distributed systems with a global clock. *Formal Methods in System Design*, 49(1-2):109–158, 2016.

[39] Christian Colombo, Gordon J Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 135–149. Springer, 2008.

[40] Christian Colombo, Gordon J Pace, and Gerardo Schneider. LARVA—safer monitoring of real-time Java programs (tool paper). In *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, pages 33–37. IEEE, 2009.

[41] Christian Colombo, Gordon J Pace, and Patrick Abela. Compensation-aware runtime monitoring. In *International Conference on Runtime Verification*, pages 214–228. Springer, 2010.

[42] Christian Colombo, Adrian Francalanza, Ruth Mizzi, and Gordon J Pace. PolyLarva: runtime verification with configurable resource-aware monitoring boundaries. In *International Conference on Software Engineering and Formal Methods*, pages 218–232. Springer, 2012.

[43] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, 2018.

[44] Ben d'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: Runtime monitoring of synchronous systems. In *Temporal Representation and Reasoning, 2005. TIME 2005. 12th International Symposium on*, pages 166–174. IEEE, 2005.

[45] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[46] Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. *International Journal on Software Tools for Technology Transfer*, 18(2):205–225, 2016.

[47] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *ACM SIGPLAN Notices*, volume 50, pages 689–700. ACM, 2015.

[48] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. *ACM SIGPLAN Notices*, 48(6): 321–332, 2013.

[49] Arvind Easwaran, Sampath Kannan, and Oleg Sokolsky. Steering of discrete event systems: Control theory approach. *Electronic Notes in Theoretical Computer Science*, 144(4):21–39, 2006.

[50] Khalil El-Harake, Ylies Falcone, Wassim Jerad, Mattieu Langet, and Mariem Mamlouk. Blocking advertisements on android devices using monitoring techniques. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 239–253. Springer, 2014.

[51] Antoine El-Hokayem and Yliès Falcone. Monitoring decentralized specifications. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 125–135. ACM, 2017.

[52] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Synthesizing enforcement monitors wrt. the safety-progress classification of properties. In *International Conference on Information Systems Security*, pages 41–55. Springer, 2008.

[53] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *International Journal on Software Tools for Technology Transfer*, 14(3): 349–382, 2012.

[54] Yliès Falcone, Tom Cornebize, and Jean-Claude Fernandez. Efficient and generalized decentralized monitoring of regular languages. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 66–83. Springer, 2014.

[55] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A stream-based specification language for network monitoring. In *International Conference on Runtime Verification*, pages 152–168. Springer, 2016.

[56] Bernd Finkbeiner, Stefan Oswald, Noemi Passing, and Maximilian Schwenger. Verified rust monitors for lola specifications. In *International Conference on Runtime Verification*, pages 431–450. Springer, 2020.

[57] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *IEEE software*, 23(2):62–70, 2006.

[58] Philip WL Fong. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 43–55. IEEE, 2004.

[59] Adrian Francalanza and Aldrin Seychell. Synthesising correct concurrent runtime monitors. *Formal Methods in System Design*, 46(3):226–261, 2015.

[60] Adrian Francalanza, Andrew Gauci, and Gordon J Pace. Distributed system contract monitoring. *The Journal of Logic and Algebraic Programming*, 82(5-7):186–215, 2013.

[61] David Garlan, S-W Cheng, A-C Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[62] MCW Geilen. On the construction of monitors for temporal logic properties. *Electronic Notes in Theoretical Computer Science*, 55(2):181–199, 2001.

[63] Dimitra Giannakopoulou and Klaus Havelund. Automata-based verification of temporal properties on running programs. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 412–416. IEEE, 2001.

[64] Felipe Gorostiaga and César Sánchez. Striver: Stream runtime verification for real-time event-streams. In *International Conference on Runtime Verification*, pages 282–298. Springer, 2018.

[65] Jean Goubault-Larrecq and Julien Olivain. A smell of ORCHIDS. In *International Workshop on Runtime Verification*, pages 1–20. Springer, 2008.

[66] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Jian Lü, and Zhendong Su. Automatic runtime recovery via error handler synthesis. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 684–695. IEEE, 2016.

[67] Sylvain Hallé and Roger Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Transactions on Services Computing*, 5(2):192–206, 2012.

[68] Kevin W Hamlen, Greg Morrisett, and Fred B Schneider. Computability classes for enforcement mechanisms. Technical report, Cornell University, 2003.

[69] Kevin W Hamlen, Greg Morrisett, and Fred B Schneider. Certified in-lined reference monitoring on. net. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 7–16. ACM, 2006.

[70] Klaus Havelund. Monitoring with data automata. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 254–273. Springer, 2014.

[71] Klaus Havelund. Rule-based runtime verification revisited. *International Journal on Software Tools for Technology Transfer*, 17(2):143–170, 2015.

[72] Klaus Havelund and Giles Reger. Specification of parametric monitors. In *Formal Modeling and Verification of Cyber-Physical Systems*, pages 151–189. Springer, 2015.

[73] Klaus Havelund and Grigore Roşu. Monitoring programs using rewriting. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 135–143. IEEE, 2001.

[74] Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 342–356. Springer, 2002.

[75] Klaus Havelund, Giles Reger, Daniel Thoma, and Eugen Zălinescu. Monitoring events that carry data. In *Lectures on Runtime Verification*, pages 61–102. Springer, 2018.

[76] Jifeng He, CAR Hoare, and Jeff W Sanders. Data refinement refined resume. In *European Symposium on Programming*, pages 187–196. Springer, 1986.

[77] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[78] Charles Antony Richard Hoare. Proof of correctness of data representations. In *Programming methodology*, pages 269–281. Springer, 1978.

[79] M Usman Iftikhar and Danny Weyns. Activforms: Active formal models for self-adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 125–134. ACM, 2014.

[80] Farnam Jahanian, Ragunathan Rajkumar, and Sitaram CV Raju. Runtime monitoring of timing constraints in distributed real-time systems. *Real-Time Systems*, 7(3):247–273, 1994.

[81] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu. JavaMOP: Efficient parametric runtime monitoring framework. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1427–1430. IEEE, 2012.

[82] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36 (1):41–50, 2003.

[83] Raphael Khoury and Nadia Tawbi. Corrective enforcement: A new paradigm of security policy enforcement by monitors. *ACM Transactions on Information and System Security (TISSEC)*, 15(2):10, 2012.

[84] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

[85] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a run-time assurance approach for Java programs. *Formal Methods in Systems Design*, 24(2):129–155, March 2004.

[86] Moonjoo Kim, Insup Lee, Usa Sammapun, Jangwoo Shin, and Oleg Sokolsky. Monitoring, checking, and steering of real-time systems. *Electronic Notes in Theoretical Computer Science*, 70(4):95–111, 2002.

[87] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal methods in system design*, 24(2):129–155, 2004.

[88] Jan Willem Klop. *Term rewriting systems*. Centrum voor Wiskunde en Informatica, 1990.

[89] Jonathan Laurent, Alwyn Goodloe, and Lee Pike. Assuring the guardians. In *Runtime Verification*, pages 87–101. Springer, 2015.

[90] Insup Lee and Susan B Davidson. Adding time to synchronous process communications. *IEEE transactions on computers*, 100(8):941–948, 1987.

[91] Insup Lee and Susan B. Davidson. A performance analysis of timed synchronous communication primitives. *IEEE Transactions on Computers*, 39(9):1117–1131, 1990.

[92] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. How good are the specs? a study of the bug-finding effectiveness of existing java api specifications. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 602–613. IEEE, 2016.

[93] Will E Leland, Murad S Taqqu, Walter Willinger, and Daniel V Wilson. On the self-similar nature of Ethernet traffic. In *ACM SIGCOMM Computer Communication Review*, volume 23, pages 183–193. ACM, 1993.

[94] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. Tessla: runtime verification of non-synchronized real-time streams. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1925–1933, 2018.

[95] Jay Ligatti and Srikar Reddy. A theory of runtime enforcement, with results. In *European Symposium on Research in Computer Security*, pages 87–100. Springer, 2010.

[96] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[97] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security (TISSEC)*, 12(3):19, 2009.

[98] Boon Thau Loo, Joseph M Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: extensible routing with declarative queries. *ACM SIGCOMM Computer Communication Review*, 35(4):289–300, 2005.

[99] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O?Neil Meredith, Traian Florin Şerbănuţă, and Grigore Roşu. RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In *International Conference on Runtime Verification*, pages 285–300. Springer, 2014.

[100] Masoud Mansouri-Samani and Morris Sloman. GEM: A generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96, 1997.

[101] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Tools for Technology Transfer*, 14(3):249–289, 2012.

[102] Robin Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[103] Stefan Mitsch and André Platzer. ModelPlex: Verified runtime validation of verified cyber-physical system models. *Formal Methods in System Design*, 49(1-2):33–74, 2016.

[104] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *International conference on verification, model checking, and abstract interpretation*, pages 41–62. Springer, 2016.

[105] Samaneh Navabpour, Yogi Joshi, Wallace Wu, Shay Berkovich, Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister. RiTHM: a tool for enabling time-triggered runtime verification for C programs. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 603–606. ACM, 2013.

[106] Sebastian Nielebock. Towards API-specific automatic program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 1010–1013. IEEE Press, 2017.

[107] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[108] Oniguruma contributors. Oniguruma. `https://github.com/kkos/oniguruma`, 2018. Accessed: 2018-03-27.

[109] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: a hard real-time runtime monitor. In *International Conference on Runtime Verification*, pages 345–359. Springer, 2010.

[110] Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet, and Omer Nguena Timo. Runtime enforcement of timed properties revisited. *Formal Methods in System Design*, 45(3):381–422, 2014.

[111] Peter J Ramadge and W Murray Wonham. Supervisory control of a class of discrete event processes. *SIAM journal on control and optimization*, 25(1):206–230, 1987.

[112] Giles Reger. *Automata based monitoring and mining of execution traces*. PhD thesis, University of Manchester, 2014.

[113] Giles Reger and David Rydeheard. From first-order temporal logic to parametric trace slicing. In *Runtime Verification*, pages 216–232. Springer, 2015.

[114] Giles Reger and David Rydeheard. From parametric trace slicing to rule systems. In *International Conference on Runtime Verification*, pages 334–352. Springer, 2018.

[115] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. MarQ: monitoring at runtime with QEA. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 596–610. Springer, 2015.

[116] Alexis Richardson et al. Introduction to RabbitMQ. *Google UK, available at http://www. rabbitmq. com/resources/google-tech-talk-final/alexis-google-rabbitmq-talk. pdf, retrieved on Mar*, 30:33, 2012.

[117] Oliviero Riganelli, Daniela Micucci, and Leonardo Mariani. Policy enforcement with proactive libraries. In *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 182–192. IEEE, 2017.

[118] Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, 2005.

[119] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM transactions on autonomous and adaptive systems (TAAS)*, 4(2):14, 2009.

[120] Usa Sammapun. *Monitoring and checking of real-time and probabilistic properties*. PhD thesis, University of Pennsylvania, 2009.

[121] César Sánchez, Gerardo Schneider, Wolfgang Ahrendt, Ezio Bartocci, Domenico Bianculli, Christian Colombo, Yliés Falcone, Adrian Francalanza, Srđan Krstić, JoHao M Lourenço, et al. A survey of challenges for runtime verification from advanced application domains (beyond software). *arXiv preprint arXiv:1811.06740*, 2018.

[122] Fred B Schneider. Enforceable security policies. Technical report, Cornell University, 1999.

[123] Joshua Schneider, David Basin, Srđan Krstić, and Dmitriy Traytel. A formally verified monitor for metric first-order temporal logic. In *International Conference on Runtime Verification*, pages 310–328. Springer, 2019.

[124] Koushik Sen, Grigore Roşu, and Gul Agha. Generating optimal linear temporal logic monitors by coinduction. In *Annual Asian Computing Science Conference*, pages 260–275. Springer, 2003.

[125] Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Roşu. Efficient decentralized monitoring of safety in distributed systems. In *Proceedings of the 26th International Conference on Software Engineering*, pages 418–427. IEEE Computer Society, 2004.

[126] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D Keromytis. Assure: automatic software self-healing using rescue points. *ACM SIGARCH Computer Architecture News*, 37(1):37–48, 2009.

[127] Richard Smith. Working draft, standard for programming language C++ n4659. *Google Inc*, pages 03–21, 2017.

[128] Hasan Sözer, Bedir Tekinerdoğan, and Mehmet Akşit. FLORA: A framework for decomposing software architecture to introduce local recovery. *Software: practice and experience*, 39 (10):869–889, 2009.

[129] Meera Sridhar and Kevin W Hamlen. Model-checking in-lined reference monitors. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 312–327. Springer, 2010.

[130] Volker Stolz and Eric Bodden. Temporal assertions using AspectJ. *Electronic Notes in Theoretical Computer Science*, 144(4):109–124, 2006.

[131] Hans Svensson and Lars-Åke Fredlund. A more accurate semantics for distributed Erlang. In *Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop*, pages 43–54, 2007.

[132] Alvaro Videla and Jason JW Williams. *RabbitMQ in action: distributed messaging for everyone*. Manning, 2012.

[133] John Wiegley and Benjamin Delaware. Using Coq to write fast and correct Haskell. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, pages 52–62. ACM, 2017.

[134] Yoriyuki Yamagata, Cyrille Artho, Masami Hagiya, Jun Inoue, Lei Ma, Yoshinori Tanabe, and Mitsuharu Yamamoto. Runtime monitoring for concurrent systems. In *International Conference on Runtime Verification*, pages 386–403. Springer, 2016.

[135] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59 (11):56–65, 2016.

[136] Teng Zhang, Peter Gebhard, and Oleg Sokolsky. SMEDL: combining synchronous and asynchronous monitoring. In *International Conference on Runtime Verification*, pages 482–490. Springer, 2016.

[137] Teng Zhang, John Wiegley, Insup Lee, and Oleg Sokolsky. Monitoring time intervals. In *International Conference on Runtime Verification*, pages 330–345. Springer, 2017.

[138] Teng Zhang, Gregory Eakman, Insup Lee, and Oleg Sokolsky. Flexible monitor deployment for runtime verification of large scale software. In *International Symposium on Leveraging Applications of Formal Methods*, pages 42–50. Springer, 2018.

[139] Teng Zhang, John Wiegley, Theophilos Giannakopoulos, Gregory Eakman, Clément Pit-Claudel, Insup Lee, and Oleg Sokolsky. Correct-by-construction implementation of runtime monitors using stepwise refinement. In *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, pages 31–49. Springer, 2018.

[140] Teng Zhang, Greg Eakman, Insup Lee, and Oleg Sokolsky. Overhead-aware deployment of runtime monitors. In *International Conference on Runtime Verification*, pages 375–381. Springer, 2019.

[141] Wenchao Zhou, Oleg Sokolsky, Boon Thau Loo, and Insup Lee. DMaC: Distributed monitoring and checking. In *International Workshop on Runtime Verification*, pages 184–201. Springer, 2009.