A Library-Based Approach to Efficient Parametric Runtime Monitoring of Java Programs

Ein programmbibliotheksbasierter Ansatz zur effizienten parametrischen Laufzeitbeobachtung von Java-Programmen

Master's Thesis by Mateusz Parzonka

March 2013

A Library-Based Approach to Efficient Parametric Runtime Monitoring of Java Programs
Ein programmbibliotheksbasierter Ansatz zur effizienten parametrischen Laufzeitbeobachtung von Java-Programmen

Vorgelegte Master-Thesis von Mateusz Parzonka

Prüfer: Eric Bodden, Ph.D.
Betreuer: Eric Bodden, Ph.D.

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 15. März 2013

____

(Mateusz Parzonka)

Abstract

Methods for the detection of specific object interactions at runtime are applied in software testing and it-security. In parametric runtime monitoring the target program is instrumented with a runtime monitor. This monitor detects patterns of object interactions by matching the program's execution trace against a parametric property. This property describes a pattern containing free variables typed by classes. During runtime this pattern is matched by binding the variables to concrete object instances.

Monitoring of parametric properties is non-trivial due to the potentially huge number of object instances. Efficient solutions for the Java platform have been implemented as language extensions and as code generators. Library-based approaches have been tried but failed to achieve an acceptable runtime overhead. In this thesis we present an efficient library for property specification and monitoring. It can be seamlessly integrated into existing tool chains allowing the comfortable and dynamic creation of runtime monitors. Expressiveness and monitoring overhead is comparable to JavaMOP, a state-of-the-art runtime monitoring framework depending on the generation of highly optimized code.

Zusammenfassung

Erkennungsmethoden für bestimmte Objektinteraktionen zur Laufzeit werden in Softwaretests und IT-Sicherheit verwendet. Bei der parametrischen Laufzeitbeobachtung wird das zu untersuchende Programm mit einem Laufzeitmonitor instrumentiert. Dieser Monitor erkennt Muster von Objektinteraktionen über den Vergleich der Ausführungshistorie des Programms mit einer parametrischen Spezifikation. Diese Spezifikation beschreibt ein Muster welches typisierte Variablen enthält. Zur Laufzeit werden den Variablen konkrete Objektinstanzen zugewiesen und das Muster geprüft.

Die Beobachtung von parametrischen Spezifikationen ist wegen der potentiell großen Menge an Objektinstanzen nicht trivial. Auf der Java-Plattform wurden effiziente Ansätze als Spracherweiterungen und Code-Generatoren implementiert. Alle bisher entwickelten programmbibliotheksbasierten Ansätze konnten keinen akzeptablen Rechenaufwand zur Laufzeit erzielen. In dieser Arbeit präsentieren wir eine effiziente Programmbibliothek zur Spezifikation und Beobachtung von parametrischen Eigenschaften. Sie kann gut in bestehende Werkzeugketten integriert werden und erlaubt die komfortable und dynamische Erzeugung von Laufzeitmonitoren. Der Ausdrucksreichtum und Rechenaufwand während der Beobachtung ist vergleichbar mit JavaMOP, einem hochmodernen Werkzeug zur Laufzeitbeobachtung, welches auf die Generierung von optimiertem Code zurückgreift.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

1 Introduction

In this chapter, we provide an introduction to runtime verification, runtime monitoring and parametric runtime monitoring. Difficulties and challenges for the development of practical solutions for parametric runtime monitoring are characterized. We motivate our approach and provide an overview over the structure of the thesis.

1.1 Runtime Verification

Runtime verification is an approach to computer system analysis. It is based on the detection of defined properties in information extracted from a running system. This is realized by observing state changes of the examined program during execution time.

A frequent motivation is to capture erroneous behavior like deadlocks or race conditions, or to disclose specific unsound interactions between particular groups of objects. Many restrictions concerning operations on objects can be enforced by the usage of programming languages with strong typing guarantees. But not all possible violations can be detected by means of strong typing, i.e., violations depending on the internal state of the involved objects. For example, writing data to a stream, although a permitted operation with compatible data types, is considered illegal when the stream is closed. Allowed operations on typed objects or inter-related object groups, when dependent on their internal states, are commonly described using the notion of *typestate* [34]. Typestate reflects how the legal operations on stateful objects change during runtime.

A large number of tools for the detection of typestate property violations have been provided by researchers originating from the field of static program analysis and formal verification, which are known as static typestate checkers. These tools try to ensure the absence of certain violations by scrutinizing the source code without actually executing the program. Based on the source code, the tools can perform structured analysis like model checking or analyze the data flow of the program. Static analysis can give strong guarantees about the conformance to a specification, since it can prove a property to be satisfied for all possible executions of the program. As a disadvantage, it can be shown that finding all possible runtime-errors in an arbitrary program is undecidable because of limitations induced by the halting problem [23]. Static analysis, therefore, has often to resort to approximative methods by maintaining an exhaustive need of processing time and memory space.

Runtime verification can be considered a lightweight verification technique which can complement heavyweight verification techniques like model checking. The guarantees given by runtime verification are considerably weaker compared to static analysis. It can only show the absence of property violations for a finite number of executions, but cannot prove the absence on all possible executions. Confidence about the level of conformance can only be gained by a sufficient number of runs of the instrumented application, though can never constitute a proof.

1.2 Runtime Monitoring

Runtime monitoring is a form of runtime verification performed by instrumenting the program with a *runtime monitor*. In its purest form, it is confined to information extracted at runtime, which is usually understood as a representation of the application's execution history. This execution history is modeled as a sequence of events generated by the implementation, where each event represents executions of specific methods or object interactions. The monitor reacts to certain patterns in this sequence, typically incrementally and efficiently at runtime (*online monitoring*), but may also work on a finite set of recorded executions (*offline monitoring*) [24].

Formally, a run of the software system may be considered as a potentially infinite word or *trace*. A concrete execution is always finite and formally a prefix of the run. When $\mathcal{L}_P$ denotes the set of valid executions in respect to a typestate property P, the runtime monitor checks wether the execution trace τ is a word in $\mathcal{L}_P$. Generally it is assumed that a correct runtime monitor is *impartial*, i.e., a trace may only be assigned to a property if all continuations of the trace also satisfy the property [24].

In case of a matching trace, the monitor can react in different ways like sending a notification or executing recovery code when a safety property is violated. This enables the development of *reflective software systems*. Such a system can monitor its own execution trace and react to observations, such that the subsequent execution is influenced by these reactions and potentially further observed and reacted upon. This is not achievable by static program analysis *per se* and extends the idea of pure runtime verification. It can be utilized as part of a software-development methodology, in which the developer specifies properties along with code to execute when validated or violated (*Monitor-Oriented Programming*, MOP [12]).

1.3 Parametric Runtime Monitoring

In *parametric runtime monitoring* both execution traces and specifications are considered *parametric*. Traces are sequences of events over objects belonging to a defined *event type*. *Parametric properties* can be thought as predicates containing free variables, quantized over all possible concrete instances of objects being bound in an event.

Example: Assume a program written in Java is using subclasses of the `Iterator` interface. The motivation for the usage of an iterator is the traversal of a collection without exposing its concrete implementation [15]. The iterator defines the two operations next and hasNext which allow to retrieve elements in the collection and allow to check if the traversal has reached the end of this view of the underlying collection. At runtime we would like to assure that no operation is performed on the null-reference, retrieved by the next operation. This intention can be specified as a violation property, which is satisfied if a concrete iterator calls two times next without hasNext called in between. This pattern can be specified by the regular expression next next with the event alphabet $\Sigma = \{\mathsf{next}, \mathsf{hasNext}\}$. Let i be a symbol denoting iterators, $e\langle i\rangle$ be an event carrying iterators, $\Lambda i.\varphi$ be a quantification over all iterators where φ contains free variables, then $\Lambda i.(\mathsf{next}\langle i\rangle\ \mathsf{next}\langle i\rangle)$ represents the property over all iterators. Let the trace of the examined program represent all executions of all next and hasNext operations for all iterators in the program. To detect a violation, we have to match all suffixes of the trace against the pattern. When a match is found, a warning can be generated in the system or recovery measures can be triggered, which can involve a reference to the iterator instance.

1.4 Challenges in Parametric Runtime Monitoring

Figure 1.1 depicts an application instrumented with a parametric runtime monitor in relation to a user of the monitoring system. As can be seen, an application instrumented with a parametric runtime monitor consists of three conceptually separated components which have to be configured by the user.

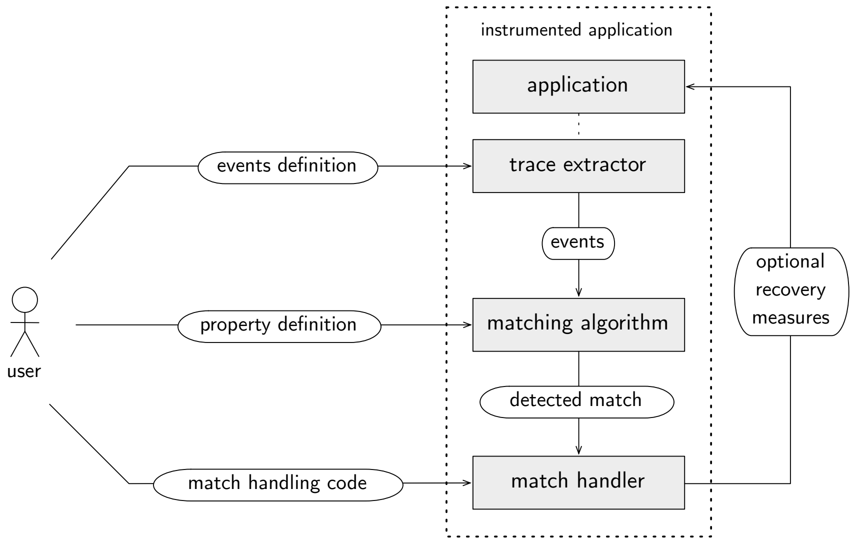


Figure 1.1.: Configuration and real-time processing in parametric runtime monitoring

Practical implementations of parametric runtime solutions have proven to be a difficult task. The challenges for a such an approach can be summarized as follows:

1. **Trace extraction** - The implementation has to disclose a trace of parametric events at runtime representing the relevant subset of the execution history of the application. It is desirable that the code of the target application has not to be changed and that monitoring code can be added unobtrusively.

2. **Property specification** - The parametric property has to be specified in an adequately rich formalism. It is desired to allow the user to resort different formalisms like regular expressions, finite-state automatons, temporal logic, context-free grammars and others. The chosen formalism and its richness plays an important role for the implementations.

3. **Trace matching** - The trace has to be matched against the specified property at runtime. Managing the large numbers of parametric events efficiently is a difficult challenge. Considering the iterator example, a runtime-monitoring approach has to keep track of every iterator created in the monitored program. More complex patterns usually require to track multiple objects per property.

4. **Match handling** - In case of a detected match, some code has to be executed. The desired effect may range between a warning and recovery measures in case of an error. Depending on the recovery measure, it can be necessary to keep a reference to the subset of objects which take part in the pattern. This further enhances the difficulty to provide a memory-efficient monitoring implementation.

5. **Ease of integration** - It is desirable that the parametric monitor is simple to create and to integrate in the runtime environment. E.g., solutions depending on language extensions or generation of optimized code are considered more difficult to integrate, since they depend on more complex platform configuration and workflow.

1.5 Motivation

The work on this thesis started with the development of lock-free data structures for the parametric monitoring library MOPBox [9]. The task was to create a prototype of a lock-free indexing scheme necessary to perform efficient monitoring and to measure its impact on performance. The insight that emerged from this project was that we first need a performant *blocking* parametric monitoring library so that synchronization can become the bottleneck. It showed that the implementation of a performant monitoring library is a challenge of its own. Efficient monitoring solutions serving as performance benchmarks exist but are either implemented as language extensions or based on generating highly optimized code. Therefore, the runtime overhead may be low but the integration of these tools into conventional Java environments may also be cumbersome. The library-based approaches known to date are easy to integrate but fail to achieve an acceptable runtime overhead. Thus, the main contribution of this thesis is our efficient library for parametric runtime monitoring on the Java platform. With the implementation of our library prm4j we can show that an library-based implementation with acceptable runtime overhead and memory consumption is possible.

1.6 Thesis Structure

In Chapter 2, we discuss important work in the field of parametric runtime monitoring. We motivate and choose an existing approach to serve as a foundation for our own solution. The theory and algorithms of Chen and Roşu [11] are presented and discussed. In Chapter 3, our approach is introduced. We discuss the algorithm and its semantics used as a basis for our implementation. Optimizations to the algorithm and the data structure are described, including the measures to prevent leaking memory. Chapter 4 contains the documentation of the implementation of the library. It provides usage examples and introduces into the structure of the project. In Chapter 5 we run a performance evaluation using an established benchmark comprising realistic target applications. The thesis closes with a discussion of future work in Chapter 6 and a conclusion in Chapter 7.

2 Background

In this chapter we present the background and the foundations for our approach. The first section comprises a selection of relevant related work. One of the presented approaches will be described in depth in the second section, as it provides the underlying theory for our approach.

2.1 Related Work

We give a short overview over important existing approaches of parametric runtime monitoring on the Java platform. Most of the approaches were selected because they are regarded as state-of-the-art. Exhaustive summaries are provided by Avgustinov et al. [2] and Bodden [8]. Initially, we describe the standard choice for trace extraction on the Java platform.

2.1.1 AspectJ

Aspect-oriented programming (AOP) is a programming paradigm. It is based on the observation that by using either procedural or object-oriented programming techniques, the implementation of some design decisions may have to be scattered throughout the code base. The resulting "tangled" code can be difficult to develop and maintain because consistent changes have to be performed at multiple places in the code. Aspect-oriented programming tackles this problem by identifying the underlying requirements as concerns which *cross-cut* the system's basic functionality. E.g., logging is considered a paradigmatic crosscutting concern, since it affects all parts of the system which have to create logs. Well-modularized abstractions of crosscutting concerns are known as *aspects*.

The AOP-implementation for the Java platform *AspectJ* [22] implements a dynamic crosscutting mechanism based on the *join point model*. In this model, join points are well-defined points in the execution of the program which can be envisioned as nodes in a simple runtime object call graph. These nodes include points at which an object receives a method call and points at which a field of an object is referenced. The edges are control flow relations between the nodes. *Pointcuts* are collections of join points used as predicates. A pointcut may match a given join point and expose information about the execution context in which the join point occurred. *Advices* declare certain code that should execute at each of the join points in a pointcut. Pointcuts, advices and ordinary Java member declarations can be comprised in aspects, forming modular units of crosscutting implementation in AspectJ.

Aspect weaving is the process of combining aspects and target code in such a way that applicable advices are run at the appropriate join points. This task can be solved either by bytecode transformation at compile time (*compile-time weaving*) or by an additional Java agent running in the JVM at load-time [17] (*load-time weaving*). A benefit in both cases is that the source of the target application does not need to be altered and aspects and application can be kept separate. The possibility to externalize code which is run at well-defined points in the execution of a target program enables a potent method for trace extraction useful in runtime monitoring. It is not surprising that this method is very common among existing runtime monitoring approaches targeting the Java platform.

2.1.2 Tracematches

Tracematches is implemented as an extension of AspectJ. It allows matching of traces modeled as sequences of predicates over join points. Parametric properties are specified as finite-state machines (FSMs) over symbols denoting those predicates. A parametric trace binds concrete parameter values to free variables used in the predicates. Matches are reported when a suffix of the current program trace is a word in the regular language specified by the FSM. This approach can be described as *state-based monitoring* [31], since a state in the FSM is associated with a monitor for each set of parameter-to-value bindings during runtime. Conceptually, it is realized by a single FSM with states labeled with constraints. Constraints are boolean expressions specified in terms of parameter-to-value bindings. E.g., a state s may be labeled with the constraint that a parameter x_1 is bound to value v_1 and parameter x_2 is bound to v_2. This expresses that the monitor for $\{x_1 \mapsto v_1, x_2 \mapsto v_2\}$ has the state s. Labels are added to states when new events are encountered. This representation conveniently models undecided or *partial* matchings as states in the FSM satisfying the constraint for a concrete group of bound objects. Thus, the same group of objects may also be associated to multiple states.

If objects of the target program need to be available in the match handler in case of a matching trace, these objects need to be referenced. Using ordinary or *strong references* pointing to objects, prolongs their lifetime preventing garbage collection which creates memory leaks. *Weak references* do not prolong object lifetime but the object may not be reachable anymore when it is needed in the match handler. Efficient management of references is, therefore, not trivial. Avgustinov et al. [2] proposes a sophisticated method to eliminate memory leaks induced by strong references to bound objects. References to objects in the monitored application can be released selectively. This happens when it can be guaranteed that a partial match will never reach a final state in the automaton. This structured release of referenced objects in case of invalidated partial matches makes tracematches one of the most efficient runtime monitoring solutions.

2.1.3 JavaMOP

JavaMOP [14] is the Java-specific instance of MOP [13], a runtime monitoring framework. JavaMOP's approach for creating efficient runtime monitors is code generation. It generates highly optimized AspectJ code for each specified monitor which has to be woven into the target program by an AspectJ compiler. The generated code is standard AspectJ code without dependencies to additional AspectJ extensions. Although JavaMOP's runtime monitors are very efficient, the reliance on code generation makes them cumbersome to integrate into existing tool chains, especially when properties have to be updated on-the-fly.

JavaMOP offers different formalisms for property specification. Users can choose from different so-called logic plugins, offering context-free grammars (CFG), past time linear temporal logic (PTLTL), FSMs and extended regular expressions (ERE) as specification formalisms. Specifications are written in a domain specific pattern language. Event definitions are provided in AspectJ and properties are written in a formalism recognized by the chosen logic plugin.

A central characteristic of JavaMOP is its formalism-independence. JavaMOP implements runtime monitoring of parametric traces by associating a non-parametric monitor to each relevant group of objects binding the free variables in the property. The concrete non-parametric monitor is considered as a black-box. During runtime, JavaMOP dispatches events to all monitors relevant for a group of objects, without making assumptions about the specific processing of those events. These events may then be processed by finite-state machines, push-down automatons or even sets of monitors allowing to match multiple suffixes of the trace at once.

JavaMOP has been carefully optimized over multiple years and has to be considered as one of the most efficient runtime monitoring solutions. As presented by Chen et al. [14] and Jin and Meredith [20], JavaMOP outperforms tracematches in terms of runtime overhead and memory consumption. However, as Bodden [8] suspects, these values may be biased due to the emphasis on property matching while neglecting management of references used in match handler code. If objects referenced in the handler are required to be available on violation or validation of a property, the weak references used in JavaMOP cannot guarantee the availability of the parameter values in case of a match. In this case JavaMOP has to fall back to strong references which causes excessive memory consumption and runtime overhead. JavaMOP's granularity regarding memory-management of bound parameter values is more coarse compared to tracematches as it does not implement a structured release of objects for failed monitors.

2.1.4 MOPBox

MOPBox [9] is an open-source library-based approach for parametric runtime monitoring on the Java platform. A MOPBox user can define runtime monitors using finite-state machines specified by API calls in Java code. AspectJ can be used for trace extraction passing parametric events as Java objects to MOPBox. MOPBox aims to be as generic as possible, granting the user flexibility in specifying the types underlying the transitions, parameters and values to be bound. Therefore, ease of integration with other libraries and environments is assured. This enables the integration with tools from domains outside of runtime verification like the Eclipse IDE[1]. Bodden [6] presents an approach to define stateful breakpoints in terms of parametric properties. A stateful breakpoint can be considered as the definition of a temporal ordering over breakpoints labeled as base events. This ordering is defined using a pattern containing free variables which can be bound to program expressions. Matching of this pattern is implemented using a parametric monitor implementation provided in MOPBox.

MOPBox itself does not provide an individual algorithmic solution for parametric runtime monitoring. It provides an abstraction over existing algorithms, so-called *indexing strategies*. The user can select concrete implementations of runtime monitoring algorithms fitting the specific property. The library contains implementations of algorithms by Chen and Roşu [11] which are also used in JavaMOP. However, as MOPBox was also developed to be used as a tool to teach runtime monitoring in an academic context, the implementations are optimized towards understandability and not performance. An official performance evaluation does not exist, however, it is confirmed by the developer of MOPBox that the runtime overhead is expected to be many magnitudes higher as compared to JavaMOP and tracematches.

[1] `http://www.eclipse.org/`

2.1.5 RV

RV [27] can be considered the closed-source successor of JavaMOP. It combines runtime monitoring with predictive analysis. The runtime monitoring component differs to JavaMOP as it features a lazy garbage collection of unnecessary monitors. A monitor is unnecessary if it will never report a match. This is determined after the collection of weakly referenced bound objects by the garbage collector has been detected. If a bound object is necessary for reaching a matching state in the monitor it is marked as collectable. RV then collects the monitor lazily employing a mark-and-sweep approach. Hence, the monitor gets removed only when its part of the data structure is accessed. This approach removes invalidated monitors after some of their weakly referenced bound objects got garbage collected. However, it does not provide a solution to the space leaks introduced by strong references as does tracematches.

RV claims to be strongly optimized. In a recent evaluation performed by Jin and Meredith [20], RV shows a runtime overhead magnitudes lower than tracematches and JavaMOP. This makes it the most efficient runtime monitoring solution to date.

2.1.6 Discussion

An ideal solution would combine the benefits from all presented approaches encapsulated as a library with a API similar to MOPBox. It would be formalism-independent like JavaMOP, provide garbage collection of monitors like RV and implement a sensible management of references to bound objects like tracematches. We, therefore, decide to chose one of the presented approaches as a base for the development of our own approach with the intention to incorporate functionality from the other approaches. Hence, we select JavaMOP and its underlying theory of *parametric trace slicing* which is described in depth in Section 2.2. The main reason is its documented low runtime overhead and intuitive event dispatching mechanism. Since the library is supposed to be used in potentially new contexts, the formalism-independence of the approach is considered a plus as it ensures future extendability.

Although formalism-independence is important, we will focus in this thesis on finite-logic specifications reducible to FSM. The reason is that real-life properties can be sufficiently specified by finite logic, as shown be a study targeting the Java platform performed by Bodden [8]. All properties encoded by linear temporal logic formalisms, as offered by JavaMOP, can be represented by alternating automata [35] which can be shown to be equivalent to finite-state machines.

Properties which can not be expressed using finite logic are properties with refer to the structure of the program. Meredith et al. [29] propose the view of finite-logic properties as *unstructured properties*, i.e., properties which do not refer to the structure of the program. These properties can only match *flat execution traces*, meaning traces without regularities. It can be shown that traces maintaining regularities which require read-write operations on a stack can not be specified sufficiently by a regular language [18]. To encode such properties, the user has to resort to context-free grammars (CFGs) or push-down automatons (PDAs). A second group of examples comprises properties in which pairs of events need to match each other, potentially in a nested way, e.g., needed to guarantee the structured locking and unlocking of resources by multiple threads. Meredith et al. show that monitoring can be performed efficiently by employing PDAs as monitors to match against patterns specified in LR(1), a subset of the deterministic context-free languages.

2.2 Foundations: Parametric Trace Slicing

In this section we will formally introduce traces, properties, monitors and a number of algorithms to perform monitoring of parametric traces without strong dependence on a specification formalism. Most definitions have been developed by Roşu and Chen [12, 13, 33, 14, 11, 32, 28] and will be referred to as *parametric trace slicing*. The definitions are often illustrated using examples centered around the *UnsafeMapIterator* property. The property specifies that a programmer may not modify a map while iterating over the map's key set or values. It is a *violation property*, implying that a match of a property has violation semantics. This is the opposite of a *validation property*, which expresses a valid state of the monitored program, as long as the property stays satisfied. Figure 2.1 displays a formal specification of the property using a deterministic finite-state machine. Such FSMs require to have a defined transition for all symbols in its alphabet. For brevity we use the convention that all edges, which are missing in the FSM, represent transitions to an invisible dead state.

The section is structured as follows: First, we will introduce parametric traces as sequences of events binding parameters to parameter values. Then we will describe the general theory of reducing a parametric trace to non-parametric traces and mapping these traces to their semantic categories. This method, called *trace slicing*, will be extended from a static calculation to efficient algorithms capable of real-time processing. The section closes with a discussion about some detected limits of the presented approach.

Note to the acquainted reader: Although most definitions are based on work of Chen and Roşu [12], some changes and extensions have been performed where seen useful. Note that some algorithms contain inlined expressions where it

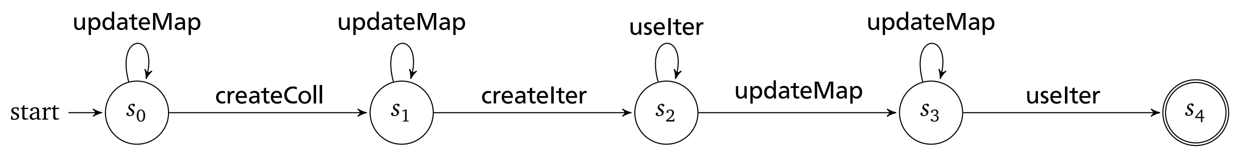


Figure 2.1.: Finite-state machine representing the UnsafeMapIterator pattern

eases reading for the unacquainted. The original notation for the most entities is kept, with some conversions and notes to achieve compatibility with the used pseudocode conventions.

2.2.1 Parametric Traces

Definition 1. ***(Base events and non-parametric traces)*** *Let $\mathcal{E}$ be the set of **base events** e. Base events are non-parametric events. In an unambiguous context they are simply called **events**. A finite sequence of events $\tau = e_1 e_2 ... e_n$ forms a **non-parametric trace**, called $\mathcal{E}$-trace or simply trace, when the context is clear. The set of all $\mathcal{E}$-traces is denoted by $\mathcal{E}^*$, letting τ be a word in $\mathcal{E}^*$.*

For example, {createColl, createIter, useIter, updateMap} is the set of events which comprises the alphabet of the FSM in Figure 2.1. createColl updateMap createIter useIter useIter and createColl createIter updateMap useIter are traces.

Definition 2. ***(Non-parametric properties and categories)*** *The semantics of non-parametric traces can be modeled by properties. A **non-parametric property** is a function $P : \mathcal{E}^* \to \mathcal{C}$, mapping each trace into the set of **categories**, expressing the relation of the trace to the property.*

Categories are a generic concept to represent semantics of the trace in regard to the property like $\{\text{violation}, \text{fail}, ?\}$ or arbitrary complex semantics like $\{\text{condition}_1, ..., \text{condition}_n\}$. The actual mapping of a trace to a category is usually defined utilizing patterns specified in a suitable formalism. A finite-state machine in form of a deterministic finite automaton is such a formalism as it can specify a language $L \in \mathcal{E}^*$ containing all relevant traces.

Example: Let ϕ_{umi} be the pattern specified by the finite-state machine in Figure 2.1 modeling a violation $\mathcal{E} = \{\text{createColl}, \text{createIter}, \text{useIter}, \text{updateMap}\}$, $\mathcal{C} = \{\text{violation}, \text{fail}, ?\}$ and $L(\phi_{\text{umi}}) \subset \mathcal{E}^*$ be the language defining all violating traces, then the property $P_{\text{umi}} : \mathcal{E}^* \to \mathcal{C}$ is defined as follows:

$$P_{\text{umi}}(\tau) = \begin{cases} \text{violation} & \textit{when } \tau \in L(\phi_{\text{umi}}) \\ \text{fail} & \textit{when no } \tau' \textit{ exists in } \mathcal{E}^* \textit{ such that } \tau\tau' \in L(\phi_{\text{umi}}) \\ ? & \textit{otherwise} \end{cases}$$

A violating trace will be mapped into the violation category: $P_{\text{umi}}(\text{createColl createIter useIter useIter updateMap useIter}) = \text{violation}$.

Non-violating traces like createColl createIter useIter useIter, which may reach the accepting state in ϕ_{umi}, are undefined in P_{umi}. The category fail can only be returned if no continuations for the trace exist, such that it will become a word in $L(\phi_{\text{umi}})$. Then the trace has no possibility to reach the violation category in the future. Considering a specification as FSM, this is always the case if a dead state is reached. E.g., if the trace starts with createColl createColl, ϕ_{umi} reaches a dead state. Note that although this is not possible in the real world, since the same collection can not be created twice from the same map, this can happen with other properties. The subset of non-failing and definite categories, sometimes called "goals", is denoted by $\mathcal{G} \subset \mathcal{C}$. We will use the notion of "matching a property" or "matching a pattern" to denote the mapping of a trace to a category in $\mathcal{G}$. In practice, $\mathcal{G}$ often comprises only one element, so the notion is usually definite.

Definition 3. ***(Parameters, parameter values and bindings)*** *Let X be the set of **parameters** and let V be the set of corresponding **parameter values**. A parameter $x \in X$ can be associated with a parameter value $v \in V$. This association is called a **binding** and modeled by a tuple in $X \times V$.*

In our running example, we define the parameter set as $X = \{m, c, i\}$, where each parameter in $\{m, c, i\}$ represents a type in {Map, Collection, Iterator}. Parameter values are concrete instances of these types, such as in $V = \{m_1, c_1, c_2, i_1, i_2, i_3\}$.

Definition 4. ***(Parameter instances)*** *An association of multiple parameters to values is modeled by a partial function $\theta : X \rightharpoonup V$ called **parameter instance**. The domain of a parameter instance is defined as $\textsc{Dom}(\theta) = \{x \mid x \in X \wedge \theta(x) \text{ defined}\}$[2]. We say an instance θ "matches" a parameter set $\mathcal{X} \subseteq X$ if $\textsc{Dom}(\theta) = \mathcal{X}$.*

[2] Not to be confused with the *domain of the definition* of the parameter instance θ, which would be $\textsc{Dom}_{def}(\theta) = X$ for all $\theta \in [X \rightharpoonup V]$.

A parameter instance can be interpreted as a set of bindings, resulting from the application of function $\mathcal{B} : [X \rightharpoonup V] \rightarrow \mathcal{P}_f(X \times V)$ with $\mathcal{B}(\theta) = \{(x, \theta(x)) \mid x \in X \wedge \theta(x) \text{ is defined}\}$. This enables a shorthand notation to represent parametric instances: $\langle x_1 \mapsto v_1, ..., x_n \mapsto v_n \rangle$ with $x_i \mapsto v_i \in \mathcal{B}(\theta)$. If the parameter-value association is notationally unambigious, the concise notation $\langle v_1, ..., v_n \rangle$ and $\langle v_1 v_2 ... v_n \rangle$ will be applied to represent instances.
Example. If θ is a parametric instance with $\theta(m) = m_1$, $\theta(c) = c_2$ and $\theta(i)$ undefined, then $\mathcal{B}(\theta) = \{(m, m_1), (c, c_2)\} = \{m \mapsto m_1, c \mapsto c_2\}$. It is easy to see that θ matches the parameter set $\{m, c\}$. Since all mappings have the form $x \mapsto x_i$, we can use the notation $\langle m_1 c_2 \rangle$ to represent θ if the symbolism is definite.

Definition 5. ***(Parametric events and event definitions)*** *A* ***parametric event*** *is a tuple (e, θ) in $\mathcal{E} \times [X \rightharpoonup V]$. We will commonly write $e\langle\theta\rangle$ to denote the parametric event and $\mathcal{E}\langle X\rangle$ to denote the set of parametric events. The* ***event definition*** *$\mathcal{D}_{\mathcal{E}} : \mathcal{E} \rightarrow \mathcal{P}_f(X)$ is a function specifying the parameter set associated to the base event. For each event $e\langle\theta\rangle \in \mathcal{E}\langle X\rangle$, $\mathcal{D}_{\mathcal{E}}(e) = \text{Dom}(\theta)$ must hold.*

Based on above notation we will specify parametric events using short-hand notation. E.g., the parametric event $e\langle x_1 \mapsto v_1, ..., x_n \mapsto v_n\rangle$ with $x_i \mapsto v_i \in \mathcal{B}(\theta)$ can be expressed with $e\langle v_1, ..., v_n\rangle$ and $e\langle v_1 v_2 ... v_n\rangle$ if the parameter-value association is definite. Example event definitions for UnsafeMapIterator: $\mathcal{D}_{\mathcal{E}}(\text{createColl}) = \{m, c\}$ and $\mathcal{D}_{\mathcal{E}}(\text{useIter}) = \{i\}$.

Definition 6. ***(Parametric traces)*** *A parametric trace τ is a word in $\mathcal{E}\langle X\rangle^*$ with ϵ being the empty trace or word.*

Assuming the set of parameters and parameter values as defined above, we specify the events createColl$\langle m \mapsto m_1, c \mapsto c_1\rangle$ and createIter$\langle c \mapsto c_1, i \mapsto i_1\rangle$. Since the used notation allows a definite interpretation of the parameter-value association, we can define a valid trace $\tau =$ createColl$\langle m_1 c_1\rangle$ createIter$\langle c_1 i_1\rangle$. Note that although non-parametric traces and parametric traces are denoted with the same symbol τ, the domain of the trace is always clear from the context.

Definition 7. ***(Compatibility and informativeness)*** *Parameter instance θ is* ***compatible*** *with parameter instance θ', written $\theta \asymp \theta'$, if for any $x \in \text{Dom}(\theta) \cap \text{Dom}(\theta')$ the equality $\theta(x) = \theta'(x)$ holds. θ is* ***less informative*** *than θ', written $\theta \sqsubseteq \theta'$, if and only if for any $x \in X$, if $\theta(x)$ is defined, then $\theta'(x)$ is also defined and $\theta(x) = \theta'(x)$. θ is* ***strictly less informative*** *than θ', written $\theta \sqsubset \theta'$, if $\theta \sqsubseteq \theta'$ and $\theta \neq \theta'$.*

Informativeness and compatibility of instances can be related, as shown in Figure 2.2. Let $X = \{x_1, ... x_4\}$ represent parameters and $V = \{v_1, ..., v_5\}$ represent parameter values. The left side of the figure shows mappings of x_i to v_j for an instance $\theta \in [X \rightharpoonup V]$, the right side of the figure represents analogous mappings defined for instance $\theta' \in [X \rightharpoonup V]$. Horizontal areas model current mappings for θ and θ' annotated by the compatibility and informativeness relations.

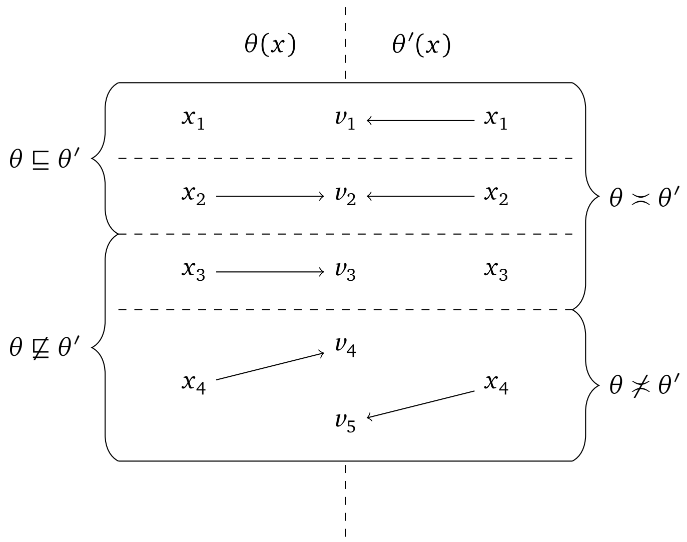


Figure 2.2.: Informativeness and compatibility of instances

The trace $\tau =$ createColl$\langle m_1 c_1\rangle$ createIter$\langle c_1 i_1\rangle$ consists of two parametric events carrying the instances $\langle m_1 c_1\rangle$ and $\langle c_1 i_1\rangle$. These instances are compatible, but no instance is less informative than the other. The parametric instance $\perp \in [X \rightharpoonup V]$ with $\perp(x)$ undefined for all $x \in X$ is compatible with and less informative than every other instance in $\theta \in [X \rightharpoonup V]$.

Definition 8. ***(Combination of instances)*** *Two compatible instances θ and θ' can be* ***combined****, written $\theta \sqcup \theta'$, resulting in the following function:*

$$(\theta \sqcup \theta')(x) = \begin{cases} \theta(x) & \textit{when } \theta(x) \textit{ is defined} \\ \theta'(x) & \textit{when } \theta'(x) \textit{ is defined} \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

Example: $\langle m_1 c_1 \rangle$ and $\langle c_1 i_1 \rangle$ are compatible and can be combined to $\langle m_1 c_1 i_1 \rangle$. $\langle m_1 c_1 \rangle$ and $\langle c_2 i_1 \rangle$ are not compatible and can not be combined.

2.2.2 Parametric Trace Slicing

Definition 9. ***(Trace slicing)*** *Let τ be a trace in $\mathcal{E}\langle X \rangle$ and θ be a parametric instance in $[X \rightharpoonup V]$. Given θ, let $\upharpoonright_\theta$ be a function in $[\mathcal{E}\langle X \rangle^* \to \mathcal{E}^*]$, written in post-fix notation, with*

$$\epsilon \upharpoonright_\theta = \epsilon$$
$$(e\langle \theta' \rangle \tau) \upharpoonright_\theta = \begin{cases} e(\tau \upharpoonright_\theta) & \textit{when } \theta' \sqsubseteq \theta \\ \tau \upharpoonright_\theta & \textit{when } \theta' \not\sqsubseteq \theta \end{cases}$$

A trace-slice $\tau \upharpoonright_\theta$ is a concatenation of non-overlapping, non-parametric subsequences of τ respecting the original ordering of its events. Intuitively, a slice for a particular parameter instance θ consists of all non-parametric events that have less informative parameter instances than θ. The approach of trace slicing is based on the observation that each parametric trace actually contains multiple non-parametric traces, one for each particular parameter instance. It can be considered a “filtered” version of τ, where an event passes the filter if the associated parameter instance maps all its parameters to the same parameter values as θ. For example: Given trace $\tau =$ createColl$\langle m_1 c_1 \rangle$ createColl$\langle m_1 c_2 \rangle$ createIter$\langle c_1 i_1 \rangle$ we can filter out events with instances binding its parameters to a specific set of parameter values: $\tau \upharpoonright_{\langle m_1 c_1 i_1 \rangle} =$ createColl createIter. The event createColl$\langle m_1 c_2 \rangle$ is ignored and a second createColl not added to the non-parametric trace.

Definition 10. ***(Parametric properties)*** *Let X be a set of parameters and V be te set of corresponding parameter values as in Definition 3. Let $P : \mathcal{E}^* \to \mathcal{C}$ be a non-parameteric property as defined in Definition 2. We define the parametric property as a function mapping from parametric traces over parametric instances to categories $\Lambda X.P : \mathcal{E}\langle X \rangle^* \to [[X \rightharpoonup V] \to \mathcal{C}]$ defined as:*

$$(\Lambda X.P)(\tau)(\theta) = P(\tau \upharpoonright_\theta)$$

$(\Lambda X.P)$ expresses a quantification over all parameters in X for which the property P holds. The function models the intuition that given an incoming parametric trace τ, an instance of P can be evaluated for each parameter instance θ, thus making all non-parametric traces in τ observable. Given $\tau =$ createColl$\langle m_1 c_1 \rangle$ createColl$\langle m_1 c_2 \rangle$ createIter$\langle c_1 i_1 \rangle$ useIter$\langle i_1 \rangle$ createIter$\langle c_2 i_2 \rangle$ updateMap$\langle m_1 \rangle$ useIter$\langle i_2 \rangle$ and property P_{umi} specified by the FSM in Figure 2.1, we can detect a violating trace among the trace slices, as illustrated in Table 2.1.

θ	$\tau \upharpoonright_\theta$	$P_{\text{umi}}(\tau \upharpoonright_\theta)$
$\langle m_1 \rangle$	updateMap	?
$\langle c_1 \rangle$	ϵ	?
$\langle i_1 \rangle$	useIter	fail
$\langle m_1 c_1 \rangle$	createColl updateMap	?
$\langle c_1 i_1 \rangle$	createIter useIter	fail
$\langle m_1 c_1 i_1 \rangle$	createColl createIter useIter updateMap	?
$\langle c_2 \rangle$	ϵ	?
$\langle i_2 \rangle$	useIter	fail
$\langle m_1 c_2 \rangle$	createColl updateMap	?
$\langle c_2 i_2 \rangle$	createIter	fail
$\langle m_1 c_2 i_2 \rangle$	createColl createIter updateMap useIter	violation

Table 2.1.: Trace slices and their category mappings

The parametric trace τ violates the property P_{umi} because the following equality holds:

$$\begin{aligned} & (\Lambda X.P_{\mathsf{umi}})(\tau)(\langle m_1 c_2 i_2\rangle) \\ = \; & P_{\mathsf{umi}}(\tau \restriction_{\langle m_1 c_2 i_2\rangle}) \\ = \; & P_{\mathsf{umi}}(\mathsf{createColl\ createIter\ updateMap\ useIter}) \\ = \; & \mathsf{violation} \end{aligned}$$

We are now able to slice a given trace by all given parametric bindings. This is performed by algorithm $\mathbb{A}\langle X\rangle$ [11]. Algorithm $\mathbb{A}\langle X\rangle$ consumes a parametric trace τ in $\mathcal{E}\langle X\rangle^*$ and returns the function $\mathbb{T}$ mapping instances to all trace slices of τ. Let MAX be a function in $[[X \rightharpoonup V] \to [X \rightharpoonup V]]$ which returns the instance θ, with the largest defined domain, in a set of instances Θ. Note that MAX(Θ) may not exist for some sets $\Theta \subset [X \rightharpoonup V]$, but, as shown by Chen and Roşu [11], it will always produce a value as used in algorithm $\mathbb{A}\langle X\rangle$. This instance is called the *most informative instance* of Θ.

```
Algorithm $\mathbb{A}\langle X\rangle$

Input:   $\tau \in \mathcal{E}\langle X\rangle^*$
Globals: $\Delta : [X \rightharpoonup V] \rightharpoonup S$
         $\Theta \subseteq [X \rightharpoonup V]$
Output:  $\mathbb{T} : [X \rightharpoonup V] \rightharpoonup \mathcal{E}^*$
Init:    $\mathbb{T} \leftarrow \perp$; $\mathbb{T}(\perp) \leftarrow \epsilon$; $\Theta \leftarrow \{\perp\}$

1: for all $e\langle\theta\rangle \in \tau$ do
2:     for all $\theta' \in \{\theta \sqcup \theta'' \mid \theta'' \in \Theta\}$ do
3:         $\theta_{max} \leftarrow$ MAX$(\{\theta'' \mid \theta'' \in \Theta \wedge \theta'' \sqsubseteq \theta'\})$
4:         $\mathbb{T}(\theta') \leftarrow \mathbb{T}(\theta_{max})e$
5:     end for
6:     $\Theta \leftarrow \Theta \cup \{\theta \sqcup \theta'' \mid \theta'' \in \Theta \wedge \theta \asymp \theta''\}$
7: end for
                                        Based on Chen and Roşu[11]
```

At initialization, $\mathbb{T}$ is undefined except for the input value $\perp$, which maps to the empty trace ϵ. The set of instances Θ is defined as the set containing $\perp$ as only parameter instance. In the outer loop of algorithm $\mathbb{A}\langle X\rangle$, the input trace is traversed from the first event to the last event. On each encountered event, the data structures $\mathbb{T}$ and Θ are updated, where DOM$(\mathbb{T}) = \Theta$ after each iteration. In the inner loop, the set of combined instances θ' is traversed, which originate from the combination of the input instance θ with all instances in Θ. For each combined instance, the algorithm computes θ_{max}, the *most knowledgable instance* related to θ'. The algorithm first calculates $\{\theta'' \mid \theta'' \in \Theta \wedge \theta'' \sqsubseteq \theta'\}$, the set of instances in Θ which are less informative than θ', then picks the maximal informative instance from this set. The most knowledgable instance θ_{max} is used to assign the slice $\mathbb{T}(\theta_{max})$ to $\mathbb{T}(\theta')$, extended by the base event e. After the inner loop in Line 6, Θ is augmented with all combined instances in the inner loop, so that Θ finally equals the domain of $\mathbb{T}$.

As can be shown, after traversing τ using data structures $\mathbb{T}$ and Θ, the equality $\mathbb{T}(\theta) = \tau \restriction_\theta$ holds for any θ in instance set Θ, which guarantees that we have stored the trace slice for every instance. The algorithm further ensures that the equality $\tau \restriction_\theta = \mathbb{T}(\text{MAX}(\{\theta' \mid \theta' \in \Theta \wedge \theta' \sqsubseteq \theta\}))$ holds for any θ in $[X \rightharpoonup V]$. This lets $\mathbb{T}$ capture the same slices that can be created by slicing τ by every instance in $[X \rightharpoonup V]$, which are all possible slices.

Definition 11. ***(Non-parametric monitors)*** *A* ***non-parametric monitor*** *M is a tuple $(S, \mathcal{E}, \mathcal{C}, s_0, \sigma, \gamma)$, where S is a set of states, $\mathcal{E}$ is a set of input events, $\mathcal{C}$ is a set of categories, s_0 is the initial state, $\sigma : S \times \mathcal{E} \to \mathcal{E}$ is the transition function and $\gamma : S \to \mathcal{C}$ is the output function. The transition function can be extended to $\sigma : S \times \mathcal{E}^* \to S$ as follows: $\sigma(s, \epsilon) = s$ and $\sigma(s, we) = \sigma(\sigma(s, w), e)$ for any $s \in S$, $e \in \mathcal{E}$ and $w \in \mathcal{E}^*$. M_P is a non-parametric monitor for property $P : \mathcal{E} \to \mathcal{C}$ if $\forall w \in \mathcal{E}^*(\gamma(\sigma(s_0, w)) = P(w))$.*

A monitor is defined as a Moore machine [30] with potentially an infinite number of states. Being an automaton, it allows to process a trace event by event, reflecting the semantics of the trace up to this point by the category σ is mapping to. Given a parametric trace τ and a monitor M_P, the category of a trace slice for a fixed $\theta \in [X \rightharpoonup V]$ is given by $\gamma(\tau \restriction_\theta)$. We say that a monitor reaches an *accepting state*, when the trace can be mapped to a verdict category. The notion of “monitor”, which is used here, targets at universality allowing different concrete monitor definitions. A common example for a possible concrete monitor is the FSM. Given a set of events $\mathcal{E}$, it can be understood as the definition of the language $\mathcal{L}_{\mathcal{E}}$, reaching its accepting state when a base trace forms a word in $\mathcal{L}$.

Definition 12. ***(Parametric monitors)*** *Given a parameter set X with a corresponding set of parameter values V and a monitor $M = (S, \mathcal{E}, \mathcal{C}, s_0, \sigma, \gamma)$ as defined in Definition 11, we define the* ***parametric monitor*** *as the monitor*

$$([[X \rightharpoonup V] \to S],\ \mathcal{E}\langle X\rangle,\ [[X \rightharpoonup V] \to \mathcal{C}],\ \Lambda X.s_0,\ \Lambda X.\sigma,\ \Lambda X.\gamma)$$

with

$$\begin{aligned} \Lambda X.s_0 &: [[X \rightharpoonup V] \to S], \\ \Lambda X.\sigma &: [[X \rightharpoonup V] \to S] \times \mathcal{E}\langle X\rangle \to [[X \rightharpoonup V] \to S], \\ \Lambda X.\gamma &: [[X \rightharpoonup V] \to S] \to [[X \rightharpoonup V] \to \mathcal{C}] \end{aligned}$$

defined as

$$\begin{aligned} (\Lambda X.s_0)(\theta) &= s_0 \\ (\Lambda X.\sigma)(\delta, e\langle\theta'\rangle)(\theta) &= \begin{cases} \sigma(\delta(\theta), e) & \textit{when } \theta' \sqsubseteq \theta \\ \delta(\theta) & \textit{when } \theta' \not\sqsubseteq \theta \end{cases} \\ (\Lambda X.\gamma)(\delta)(\theta) &= \gamma(\delta(\theta)) \end{aligned}$$

for any $\delta \in [[X \rightharpoonup V] \to S]$ and any $\theta, \theta' \in [X \rightharpoonup V]$.

Compared to the non-parametric monitor, all components of the monitor are functions, which have instances as parameters. States in the parametric monitor $\Lambda X.M$, including the initial state, are functions $\delta \in [[X \rightharpoonup V] \to S]$ mapping parametric instances to states of the underlying non-parametric monitor M, whereas categories in $\Lambda X.M$ map from instances to categories in M. In other words, the parametric monitor $\Lambda X.M$ maintains one state of M and one category of M per parameter instance. Therefore, a parametric monitor can be envisioned as multiple non-parametric monitors running in parallel, where each tries to match a trace slice. All these base monitors are parametrized in the same way, maintaining its own state but matching the same property.

Algorithm $\mathbb{B}\langle X\rangle(M = (S, \mathcal{E}, \mathcal{C}, s_0, \sigma, \gamma))$

```
Input:   $\tau \in \mathcal{E}\langle X\rangle^*$
Globals: $\Delta : [X \rightharpoonup V] \to S$
         $\Theta \subseteq [X \rightharpoonup V]$
Output:  $\Gamma : [X \rightharpoonup V] \to \mathcal{C}$
Init:    $\Delta \leftarrow \bot$; $\Delta(\bot) \leftarrow s_0$; $\Theta \leftarrow \{\bot\}$

1: for all $e\langle\theta\rangle \in \tau$ do
2:     for all $\theta' \in \{\theta \sqcup \theta'' \mid \theta'' \in \Theta\}$ do
3:         $\theta_{max} \leftarrow \textsc{Max}(\{\theta'' \mid \theta'' \in \Theta \wedge \theta'' \sqsubseteq \theta'\})$
4:         $\Delta(\theta') \leftarrow \sigma(\Delta(\theta_{max}), e)$
5:         $\Gamma(\theta') \leftarrow \gamma(\Delta(\theta'))$
6:     end for
7:     $\Theta \leftarrow \Theta \cup \{\theta \sqcup \theta'' \mid \theta'' \in \Theta \wedge \theta \asymp \theta''\}$
8: end for
```

Based on Chen and Roşu[11]

Chen and Roşu extend algorithm $\mathbb{A}\langle X\rangle$ by introducing monitors which process trace slices: Algorithm $\mathbb{B}\langle X\rangle(M)$ is very similar to algorithm $\mathbb{A}\langle X\rangle$, with the main difference that it depends on a given monitor M. The trace slices in algorithm $\mathbb{A}\langle X\rangle$ are replaced by multiple instances of the base monitor M processing those slices. Instead of keeping track of slices in the data structure, a state is stored for each relevant instance θ' reflecting the monitor state after processing $\tau \restriction_{\theta'}$ in the monitor M, starting from the initial state. The function Δ is the substitute for $\mathbb{T}$, mapping instances to states in M. Δ is initially only defined for $\bot$, which maps to the initial state s_0. As in algorithm $\mathbb{A}\langle X\rangle$, Θ comprises all relevant instances and is equal to the domain of Δ. Outer loop and line 7 is identical to algorithm $\mathbb{A}\langle X\rangle$. The inner loop traverses the set of combined instances, created by combining the input instance θ with all elements in Θ. In each iteration, the most knowledgable instance in Θ, in relation to the combined instance θ', is calculated. It is used as basis for an e-transition to the next state calculated by the transition function σ. The next state is then stored in the data structure Δ. The function Γ keeps track of the semantics for each trace slice by storing the the category value $\gamma(s)$ for each state s in the co-domain of Δ. If the category is a verdict category, then it is possible to output a message including θ'. After processing input trace τ, the state stored in Δ equals the state reached after a slice $\tau \restriction_\theta$ is processed in the underlying base monitor, i.e., $\Delta(\theta') = \sigma(s_0, \tau \restriction_{\theta'})$. It can be concluded with Definition 12 that an instance of $\mathbb{B}\langle X\rangle(M)$ is a parametric monitor $\Lambda X.M$, where monitor M is a monitor for property P.

Algorithm $\mathbb{C}\langle X\rangle(M = (S, \mathcal{E}, \mathcal{C}, s_0, \sigma, \gamma))$

```
Globals: $\Delta : [X \rightharpoonup V] \rightharpoonup S$
         $\mathcal{U} : [X \rightharpoonup V] \rightarrow \mathcal{P}_f([X \rightharpoonup V])$
Init:    $\Delta(\bot) \leftarrow s_0$
         $\mathcal{U}(\theta) \leftarrow \emptyset$ for any $\theta \in [X \rightharpoonup V]$
Output:  $\Gamma : [X \rightharpoonup V] \rightharpoonup \mathcal{C}$

1:  function MAIN($e\langle\theta\rangle$)
2:      if $\Delta(\theta)$ undefined then
3:          for all $\theta_{max} \sqsubset \theta$ (in reversed topological order) do
4:              if $\Delta(\theta_{max})$ defined then
5:                  go to 8
6:              end if
7:          end for
8:          DEFINETO($\theta, \theta_{max}$)
9:          for all $\theta_{max} \sqsubset \theta$ (in reversed topological order) do
10:             for all $\theta_{comp} \in \mathcal{U}(\theta_{max})$ s.t. $\theta_{comp}$ is compatible with $\theta$ do
11:                 if $\Delta(\theta_{comp}) \sqcup \theta$ undefined then
12:                     DEFINETO($\theta_{comp} \sqcup \theta,\ \theta_{comp}$)
13:                 end if
14:             end for
15:         end for
16:     end if
17:     for all $\theta' \in \mathcal{U}(\theta) \cup \{\theta\}$ do
18:         $\Delta(\theta') \leftarrow \sigma(\Delta(\theta), e)$
19:         $\Gamma(\theta') \leftarrow \gamma(\Delta(\theta'))$
20:     end for
21: end function

22: function DEFINETO($\theta, \theta'$)
23:     $\Delta(\theta) \leftarrow \Delta(\theta')$
24:     for all $\theta'' \sqsubset \theta$ do
25:         $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$
26:     end for
27: end function
```

Reproduced from Chen and Roşu[11]

Algorithm $\mathbb{B}\langle X\rangle$ is not very efficient. The set Θ maintaining all combinations of parameter instances, which are observed in parametric trace events, gets very large over time. For each incoming event its input instance has to be combined with all instances in this set, after which, in the inner loop, all less informative instances have to be selected. These expensive operations would yield an excessive runtime overhead in real-life situations.

Chen and Roşu [11], therefore, propose the optimized algorithm $\mathbb{C}\langle X\rangle$ to perform efficient online monitoring. The algorithm is parametrized with a non-parametric monitor, as defined in Definition 11, and allows to process a potentially infinite parametric trace event by event.

The algorithm maintains two data structures: Δ is a partial mapping from the set of instances to the set of states introduced by the non-parametric monitor. Its purpose is the same as in algorithm $\mathbb{B}\langle X\rangle$, namely to map an relevant instance θ to a monitor state, representing the state of the trace slice $\tau\restriction_\theta$. The Γ function is defined identically and works as expected, mapping $[X \rightharpoonup V]$ to categories, which reflect the semantics of the slices. A major difference to algorithm $\mathbb{B}\langle X\rangle$ is the missing set of relevant instances Θ, which is not explicitly maintained in algorithm $\mathbb{C}\langle X\rangle$. Instead, a function $\mathcal{U}$ is introduced, which maps each possible instance θ to a set of relevant instances which are strictly more informative than θ. The equality $\mathcal{U}(\theta) = \{\theta' \in \text{DOM}(\Delta) \mid \theta \sqsubset \theta'\}$ holds after each processed event.

At initialization, the partial function Δ is defined only for $\bot$, mapping to the initial state introduced by the non-parametric monitor M. $\mathcal{U}$ is initially defined for possible instances, mapping each instance to the empty set of instances.

$$\begin{tikzcd}
 & \bot \arrow[dl] \\
\langle a_1 \rangle & \\
 & \langle a_1 b_1 \rangle
\end{tikzcd}$$

(a) Incoming instance $\langle a_1 b_1 \rangle$

$$\begin{tikzcd}
 & \bot \arrow[dl] \\
\langle a_1 \rangle & \\
 & \langle a_1 b_1 \rangle \arrow[ul, dashed]
\end{tikzcd}$$

(b) Find $\theta_{max} = \langle a_1 \rangle$

$$\begin{tikzcd}
 & \bot \arrow[dl] \\
\langle a_1 \rangle \arrow[dr, dashed, "\Delta(\langle a_1 \rangle)"'] & \\
 & \langle a_1 b_1 \rangle
\end{tikzcd}$$

(c) Copy monitor state

$$\begin{tikzcd}
 & \bot \arrow[dl] \arrow[dr] \arrow[dd] & \\
\langle a_1 \rangle \arrow[dr] & & \langle b_1 \rangle \arrow[dl] \\
 & \langle a_1 b_1 \rangle &
\end{tikzcd}$$

(d) Update $\mathcal{U}$-function

Figure 2.3.: Find-max phase

The algorithm consists of the two functions MAIN and DEFINETO. DEFINETO takes two parameter instances $\theta, \theta' \in [X \rightharpoonup V]$ as arguments. When the function is called, θ' has a defined value in Δ, but $\Delta(\theta)$ is undefined. In Line 23, the value of $\Delta(\theta')$ is copied and defined at $\Delta(\theta)$. The function now adds θ to all instance sets in the co-domain of $\mathcal{U}$ that are associated to instances strictly less informative than θ, which ensures that $\mathcal{U}(\theta) = \{\theta' \in \mathrm{DOM}(\Delta) \mid \theta \sqsubset \theta'\}$. Note that usually $\mathrm{DOM}(\Delta) \neq \mathrm{DOM}(\mathcal{U})$, because $\mathcal{U}$ is defined in respect to all possible instances which may be encountered in the future, therefore, not (yet) defined in Δ.

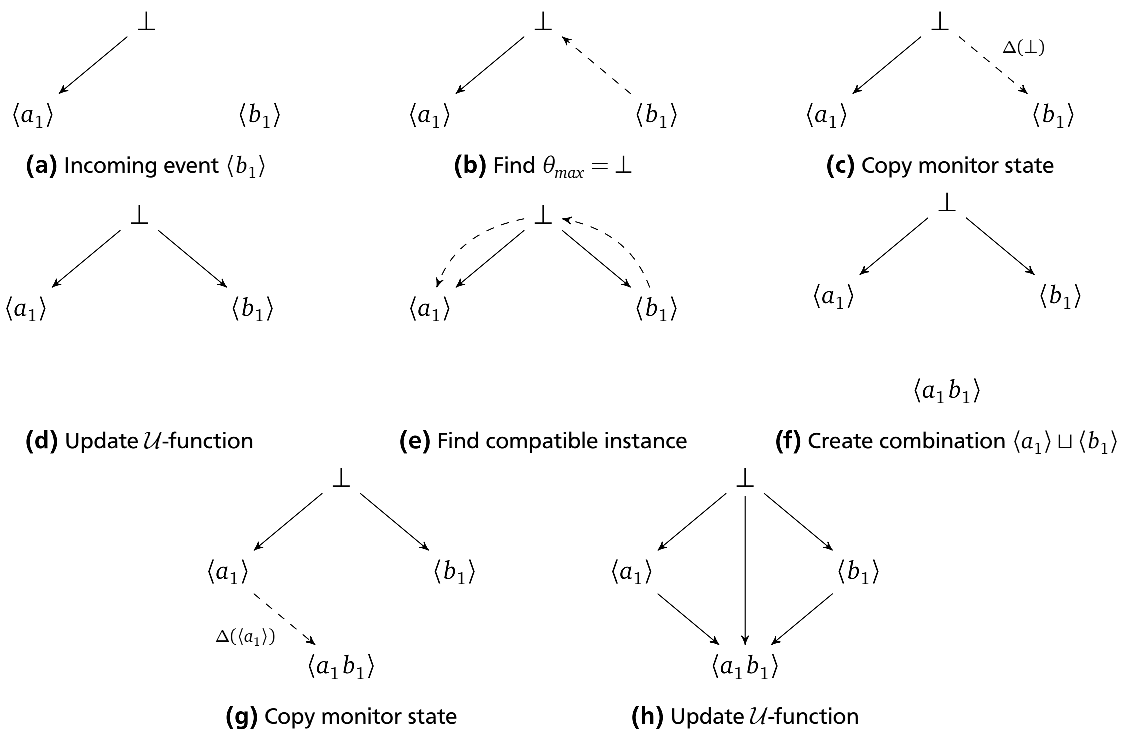
MAIN takes a parametric event $e\langle\theta\rangle$ as argument, consuming a parametric trace on event-to-event basis. The function differentiates two cases, depending on whether an incoming instance is already known or yet unknown. It can be divided into three distinguishable phases, which we named *find-max* (Lines 3-8), *join* (Lines 9-15) and *update phase* (Lines 17-20).

The simple case is, when an event $e\langle\theta\rangle$ carries an known instance, meaning an instance which is already defined in Δ. The algorithm then skips the find-max and join phase and enters the update phase. In this phase, both Δ and Γ are updated for all instances for which the incoming base event is relevant, i.e., the set of strictly more informative instances including the input instance $\{\theta' \in \mathrm{DOM}(\Delta) \mid \theta \sqsubset \theta'\} \cup \theta$.

In the second case, when the incoming instance θ is initially unknown, the algorithm enters the find-max phase followed by the join phase. It first traverses all strictly less informative instances to find the most informative instance already defined in Δ. Note that this search happens in reverse topological order, meaning from the most informative instance to the least informative instance, and stops after θ_{max} is found. The least informative instance is $\bot$, which was defined at initialization, so the search for θ_{max} always succeeds. The find-max phase ends, when DEFINETO is called, setting the monitor state for the new instance θ to the state of θ_{max} and updating $\mathcal{U}$ accordingly. In the join phase, the algorithm traverses all strictly less informative instances θ_{max} again, this time not stopping on the first defined instance, and iterates through all instances θ_{comp} which are strictly more informative than θ_{max} and compatible with the input instance θ. The compatibility check is needed to ensure that the combination of θ and θ_{comp} exists. $\Delta(\theta_{comp} \sqcup \theta)$ is set to $\Delta(\theta_{comp})$ if still undefined and $\mathcal{U}$ is updated accordingly. After the join phase, $\Delta(\theta)$ is defined and the algorithm proceeds with the update phase as described in the simple case.

Figure 2.3 visualizes the $\mathcal{U}$-function as a directed graph, where nodes are instances and edges represent a mapping between a source and a destination, e.g., $\mathcal{U}(\theta_{source}) = \{\ldots, \theta_{dest}, \ldots\}$. All edges connect nodes from depth i to depth $i+1$, where each depth i contains only nodes representing instances which map i parameters to parameter values. Note that the graph is no arborescence, since there may exist multiple paths to a node with depth $i > 1$.

Figure 2.3 shows a visualization of calculation steps of algorithm $\mathbb{C}\langle X\rangle$, which is based on a graph representing the $\mathcal{U}$-function. The instances are named following a short-hand naming scheme where a parameter x maps to a parameter value x_i, i.e., $\langle a \mapsto a_1, b \mapsto b_1\rangle = \langle a_1 b_1\rangle$. Solid edges denote an $\mathcal{U}$-relation, where dashed edges are used to represent relationships between instances used in some calculation steps. The current execution state of the algorithm is depicted in Figure 2.3a, showing a known instance $\langle a_1\rangle$ and an incoming input instance $\langle a_1 b_1\rangle$, symbolized by the unconnected node. The dashed line in the Subfigure 2.3b represents the search for an already defined θ_{max}, which is found in $\langle a_1\rangle$. Note that although the $\mathcal{U}$-relation equates to the $\sqsubset$-relation for all known instances, the set of instances in $\{\theta' \mathrm{DOM}(\Delta) \mid \theta' \sqsubset \theta\}$ has to calculated for each new instance θ beforehand. In Figure 2.3c, the monitor state from $\langle a_1\rangle$ is used to define $\Delta(\langle a_1 b_1\rangle)$. In the last shown step, a mapping for each known instance which is strictly less informative than $\langle a_1 b_1\rangle$ is added to $\mathcal{U}$.



(a) Incoming event $\langle b_1 \rangle$ **(b)** Find $\theta_{max} = \perp$ **(c)** Copy monitor state **(d)** Update $\mathcal{U}$-function **(e)** Find compatible instance **(f)** Create combination $\langle a_1 \rangle \sqcup \langle b_1 \rangle$ **(g)** Copy monitor state **(h)** Update $\mathcal{U}$-function

Figure 2.4.: Find-max phase followed by join phase

Figure 2.4 visualizes the computation steps involving a find-max phase and a combination phase. Steps a-d show the find-max phase, which is analogous to the example already discussed. In Figure 2.4e, the combination phase starts. The dashed lines show the search path over the less informative instance $\perp$ to the compatible instance $\langle a_1 \rangle$. A new combination is created, which receives the monitor state from the compatible instance. Note that in some cases a combination may be created, which is already defined in Δ, e.g., $\theta = \langle b_1 c_1 \rangle$ the new instance, and $\langle a_1 \rangle$ and $\langle a_1 b_1 \rangle$ existing compatible instances. It is, therefore, crucial that the strictly less informative instances are descended from most informative to least informative, to guarantee that the new instance is combined with the most informative compatible instance ($\langle a_1 b_1 \rangle$) first. Subfigure 2.4h closes with the final update of the $\mathcal{U}$-function.

2.2.4 Efficient Online Parametric Trace Slicing

As the intuition of the core approach of efficient parametric monitoring is now clear, we will define a notion to discuss the computations in the algorithms more concisely.

Definition 13. ***(Instance monitors and monitor instances)*** *Let $\Delta : [X \rightharpoonup V] \to S$ be a function mapping instances $\theta \in [X \rightharpoonup V]$ to states $s \in S$. If a mapping $\Delta(\theta) = s$ exists, then θ is "known" and this mapping is called* ***instance monitor*** *or simply monitor if the context is clear. Since the mapping is bijective, each monitor state is associated with exactly one instance, where such θ is called* ***monitor instance****.*

If a mapping $\Delta(\theta)$ is defined, then we say that instance θ "has a monitor". If at some point in the computation $\Delta(\theta)$ is undefined and after an algorithmic step, $\Delta(\theta)$ is defined, the monitor for instance θ has been "created". When a mapping $\Delta(\theta)$ is defined at some point and an assignment $\Delta(\theta) \leftarrow s$ is performed at a later step in the computation, we say that the monitor for θ was "updated".

Algorithm $\mathbb{C}\langle X \rangle$ optimizes the search for the most knowledgable instance by handling two cases of existing compatible instances. An efficient instance lookup is a first target for optimization. Other targets are related to monitor creation, maintenance and updates. It is desired that only monitors are created and updated which still have the chance to reach an accepting state. If a monitor has no possibility of reaching an accepting state, the creation would be a waste of resources and any update would be in vain.

As can be shown [14], not all monitors created by algorithm $\mathbb{C}\langle X \rangle$ are needed to ensure that all slices matching properties are detected. One example can be given by property P_ϕ formalized by the regular expression $\phi = {e_2}^* e_1 e_2$, where each event e_i carries an instance $\langle x \mapsto x_j \rangle$, i.e., $\mathcal{D}_\mathcal{E}(e_i) = \{x\}$. Let us assume that the event $e_2\langle x_1 \rangle$ is passed to the algorithm and the instance $\langle x_1 \rangle$ is unknown up to this point.

Algorithm $\mathbb{C}^+\langle X\rangle(M = (\mathcal{S}, \mathcal{E}, \mathcal{C}, s_0, \sigma, \gamma))$

Globals: $\Delta : [X \rightharpoonup V] \rightharpoonup \mathcal{S}$
$\mathcal{U} : [X \rightharpoonup V] \rightarrow \mathcal{P}_f([X \rightharpoonup V])$
Init: $\mathcal{U}(\theta) \leftarrow \emptyset$ for any $\theta \in [X \rightharpoonup V]$
Output: $\Gamma : [X \rightharpoonup V] \rightharpoonup \mathcal{C}$

```
 1: function MAIN($e\langle\theta\rangle$)
 2:     if $\Delta(\theta)$ undefined then
 3:         for all $\theta_{max} \sqsubset \theta$ (in reversed topological order) do
 4:             if $\Delta(\theta_{max})$ defined then
 5:                 go to 8
 6:             end if
 7:         end for
 8:         if $\Delta(\theta)$ defined then
 9:             DEFINETO($\theta, \theta_{max}$)
10:         else if $e$ is a creation event then
11:             DEFINENEW($\theta$)
12:         end if
13:         DEFINETO($\theta, \theta_{max}$)
14:         for all $\theta_{max} \sqsubset \theta$ (in reversed topological order) do
15:             for all $\theta_{comp} \in \mathcal{U}(\theta_{max})$ s.t. $\theta_{comp}$ is compatible with $\theta$ do
16:                 if $\Delta(\theta_{comp}) \sqcup \theta$ undefined then
17:                     DEFINETO($\theta_{comp} \sqcup \theta,\ \theta_{comp}$)
18:                 end if
19:             end for
20:         end for
21:     end if
22:     for all $\theta' \in \mathcal{U}(\theta) \cup \{\theta\}$ do
23:         $\Delta(\theta') \leftarrow \sigma(\Delta(\theta), e)$
24:         $\Gamma(\theta') \leftarrow \gamma(\Delta(\theta'))$
25:     end for
26: end function

27: function DEFINENEW($\theta$)
28:     $\Delta(\theta) \leftarrow s_0$
29:     for all $\theta'' \sqsubset \theta$ do
30:         $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$
31:     end for
32: end function

33: function DEFINETO($\theta, \theta'$)
34:     $\Delta(\theta) \leftarrow \Delta(\theta')$
35:     for all $\theta'' \sqsubset \theta$ do
36:         $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$
37:     end for
38: end function
```

Reproduced from Chen et al.[14]

The mapping $\Delta(\langle x_1\rangle) = s_0$ is created without adding any useful information. Since the state associated to $\langle x_1\rangle$ is still the initial state, it does not matter for any of the possible future e_1 events how many times e_2 was processed before by the instance monitor for $\langle x_1\rangle$. A more efficient solution would simply ignore any $e_2\langle x_i\rangle$ events until an $e_1\langle x_i\rangle$ event is encountered. To address this inefficiency, Chen et al. propose the enhanced algorithm $\mathbb{C}^+\langle X\rangle$ [14] (see page 15) which tests if the input event is a *creation event* before creating a new monitor.

Note that this approach demands knowledge about creation events given before execution of the algorithm. It is known from the implementation of JavaMOP that creation events can be configured by the user as part of the property specification. This has consequences on the semantics of JavaMOP which are discussed in Section 3.1.1.1.

The creation event optimization prevents needless instantiation of monitors in initial state. But there are other situations in which monitors are created without constituting the correctness of the approach. As described in our short summery of the workings of algorithm $\mathbb{C}\langle X\rangle$, we identified two parts in the algorithm where new monitors are instantiated reusing the state of an existing monitor. In the join phase, input instances are combined with known compatible instances. This happens even if the instance monitors of the combinations will never reach a verdict category. Algorithm $\mathbb{C}^+\langle X\rangle$ is not aware of the specified property or the semantics of the observed program so it can not determine that the monitor creation is unnecessary in this case.

Table 2.2 shows Δ after consuming the input trace createColl$\langle m_1c_1\rangle$ useIter$\langle i_2\rangle$.

$e\langle\theta\rangle$	Δ	Γ
createColl$\langle m_1c_1\rangle$	$\langle m_1c_1\rangle \mapsto s_1$	$\langle m_1c_1\rangle \mapsto$?
useIter$\langle i_2\rangle$	$\langle m_1c_1\rangle \mapsto s_1$	$\langle m_1c_1\rangle \mapsto$?
	$\langle m_1c_1i_2\rangle \mapsto \sigma(\Delta(\langle m_1c_1\rangle),$ useIter$)$	$\langle m_1c_1i_2\rangle \mapsto$ fail

Table 2.2.: $\mathbb{C}^+\langle X\rangle$ with input trace createColl$\langle m_1c_1\rangle$ useIter$\langle i_2\rangle$

The instance $\langle m_1c_1\rangle$ is unknown and createColl is a creation event, so a monitor with initial state is created for $\langle m_1c_1\rangle$. The next event carries an unknown instance $\langle i_2\rangle$ which is compatible with $\langle m_1c_1\rangle$, thus an instance monitor for the combination $\langle m_1c_1i_2\rangle$ is created. The new instance monitor is associated with the state $\sigma(\Delta(\langle m_1c_1\rangle),$ useIter$)$, which evaluates to the dead state, implying the monitor will never report a verdict category.

The intuition, why this is inefficient, is based on knowledge of the property. The violation pattern, as defined in the FSM in Figure 2.5, specifies creation dependencies between collections, maps and iterators. This formalizes that collections are created by maps, and iterators are created by collections, implying that a collection has only one map and an iterator has only one collection.

The FSM is a definition of base events which label transitions on a path that will eventually reach the accepting state. Some events may only be part of the valid sequence when triggered from certain states. E.g., useIter leads to the dead state from s_0 and s_1, loops on s_2, but is necessary to reach the accepting state with a transition from s_3. This specifies that an iterator may only be used with (useIter) when created from its underlying collection (createIter), which was created based on an existing map (createColl) respectively.

The dependency on preceding event occurrences can be defined as follows [14]:

Definition 14. ***(Trace enable set)*** *Given trace $\tau \in \mathcal{E}^*$ and events $e, e' \in \tau$, we denote that e' occurs before the first occurrence of e in τ as $e' \rightsquigarrow_\tau e$. Let the trace enable set of $e \in \mathcal{E}$ be the function* $\textsc{enable} : (\mathcal{E}^* \times \mathcal{E}) \to \mathcal{P}_f(\mathcal{E})$*, defined as:* $\textsc{Enable}(\tau, e) = \{e' \mid e' \in \mathcal{E} \wedge e' \rightsquigarrow_\tau e\}$.

The notion of "enabling" implies that e is assumed to be dependent on the preceding events, in the sense they "enable" its capability of occurrence. The enable set can be extended from an arbitrary trace to a trace which can be recognized to be a part of a verdict category of a property P as discussed above.

Definition 15. ***(Property event enable set)*** *Given $P : \mathcal{E}^* \to \mathcal{C}$ and a set of categories specified as goal $\mathcal{G} \subseteq \mathcal{C}$, the property event enable set is defined as the function* $\textsc{Enable}^{\mathcal{E}}_{\mathcal{G}} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$ *with* $\textsc{Enable}^{\mathcal{E}}_{\mathcal{G}}(e) = \{\textsc{Enable}(\tau, e) \mid \tau \in \mathcal{E}^* \wedge P(\tau) \in \mathcal{G}\}$.

By applying the concept of property enable sets on the pattern specified by a FSM, the traces $\tau \in \Sigma^*$ with $P(\tau) \in \mathcal{G}$ are all paths, which lead from the initial state to the accepting state associated with a verdict category in $\mathcal{G}$. In P_{smi}, all paths traverse the edges createColl, createIter and useIter in the same order, which implies that createColl enables createIter, which enables useIter. In the example stated in Table 2.2, the decision if instance $\langle i_2\rangle$ should be combined with $\langle m_1c_1\rangle$ must be derived from information that is available at this point. It is known that a monitor for $\langle m_1c_1\rangle$ exists and to which state it is mapping to. The enable set optimization proposed by Chen et al. considers only the fact that the monitor is defined. An existing instance monitor θ implies that there must have been events which led to creation of this monitor. Considering the pattern constituting the property, a necessary monitor for $\langle m_1c_1\rangle$ can only be created by an input event createColl$\langle m_1c_1\rangle$ eventually followed by any number of updateMap events. This can be derived from the parameters which are associated with the base events. As shown in Figure 2.5, each of the events introduces a specific set of parameters on its first occurrence, e.g., createColl introduces $\{m, c\}$ and createIter introduces $\{c, i\}$.

The definition of the property enable set can easily be extended to parameters instead of base events. In this case, the event is enabled by first occurrences of parameters associated with events.

Definition 16. ***(Property parameter enable set)*** *Given a property $P : \mathcal{E}^* \to \mathcal{C}$, a set of verdict categories $\mathcal{G} \subseteq \mathcal{C}$, a set of parameters X, the property parameter enable set of event $e \in \mathcal{E}$ is defined as the function* $\textsc{Enable}^{X}_{\mathcal{G}} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(X))$ *with* $\textsc{Enable}^{X}_{\mathcal{G}}(e) = \{\cup\{\mathcal{D}_{\mathcal{E}}(e') \mid e' \in \textsc{Enable}(\tau, e)\} \mid \tau \in \mathcal{E}^* \wedge P(\tau) \in \mathcal{G}\}$.

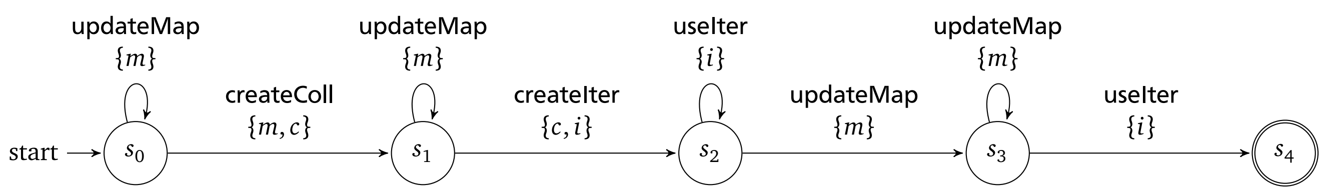


Figure 2.5.: Finite-state machine representing the UnsafeMapIterator pattern with parameter sets

Enable sets can be calculated with algorithm $\textsc{CalcEnableSets}_{\text{fsm}}$. Table 2.3 shows the result of the calculation for UnsafeMapIterator. With property parameter enable sets, the creation of unnecessary monitors can be avoided, as was shown by Chen et al. If algorithm $\mathbb{C}^{+}\langle X\rangle$ receives an event $e\langle\theta\rangle$, and a compatible instance θ' is used to define $\theta \sqcup \theta'$ and $\textsc{Dom}(\theta') \notin \textsc{Enable}_{\mathcal{G}}^{X}(e)$, then no accepting state will be reached from $\Delta(\theta \sqcup \theta')$ during the whole monitoring process. An improved algorithm, therefore, can omit the definition of $\Delta(\theta \sqcup \theta')$.

e	$\textsc{Enable}_{\mathcal{G}}^{\mathcal{E}}(e)$	$\textsc{Enable}_{\mathcal{G}}^{X}(e)$
createColl	$\{\emptyset\}$	$\{\emptyset\}$
createIter	{{createColl}, {createColl, updateMap}}	$\{\{m,c\}\}$
useIter	{{createColl, createIter}, {createColl, updateMap, createIter}}	$\{\{m,c,i\}\}$
updateMap	{{createColl}, {createColl, createIter}, {createColl, updateMap, createIter}}	$\{\{m,c\},\{m,c,i\}\}$

Reproduced from Chen et al.[14]

Table 2.3.: Property enable sets for UnsafeMapIterator

Using enable sets instead of all less informative instances suppresses the creation of monitors but also introduces a loss of information. Unfortunately, without additional precautions, it has an impact on correctness, as false positives, i.e., traces erroneously mapped to verdict categories are detected. Assume $\mathcal{E} = \{e_1, e_2, e_3\}$, $X = \{a, b\}$ and a property specified by regular expression $e_1 e_2$. e_1 is annotated as creation event, and $\mathcal{D}_{\mathcal{E}} = \{e_1 \mapsto \{a\}, e_2 \mapsto \{a,b\}, b \mapsto \{x_2\}\}$. The property parameter enable set is calculated with $\textsc{Enable}_{\mathcal{G}}^{X} = \{e_1 \mapsto \{\emptyset\}, e_2 \mapsto \{\{a\}\}, e_3 \mapsto \{\emptyset\}\}$. Consider the trace $e_1\langle a_1\rangle$ $e_3\langle b_1\rangle$ $e_2\langle a_1 b_1\rangle$. Although this trace does not match the pattern, it is matched by algorithm $\mathbb{C}^{+}\langle X\rangle$ relying on enable sets, as depicted in Table 2.4.

$e\langle\theta\rangle$	Δ	Γ
$e_1\langle a_1\rangle$	$\langle a_1\rangle \mapsto \sigma(s_0, e_1)$	$\langle a_1\rangle \mapsto$?
$e_3\langle b_1\rangle$	$\langle a_1\rangle \mapsto \sigma(s_0, e_1)$	$\langle a_1\rangle \mapsto$?
$e_2\langle a_1 b_1\rangle$	$\langle a_1\rangle \mapsto \sigma(s_0, e_1)$	$\langle a_1\rangle \mapsto$?
	$\langle a_1 b_1\rangle \mapsto \sigma(\sigma(s_0, e_1), e_2)$	$\langle a_1 b_1\rangle \mapsto$ match

Table 2.4.: Algorithm $\mathbb{C}^{+}\langle X\rangle$ considering enable sets and trace $e_1\langle a_1\rangle, e_3\langle b_1\rangle, e_2\langle a_1 b_1\rangle$ (e_1 is a creation event)

$e_3\langle b_1\rangle$ has no impact on Δ, since it is no creation event and its enable set is empty, forcing no combinations with existing instances like $\langle a_1\rangle$. On event $e_2\langle a_1 b_1\rangle$, there is no information available to detect the interleaving of e_3. As Chen et al. propose, this can be fixed using timestamps for instances. Let the function $\mathcal{T}_{\Delta}$ and $\mathcal{T}_{\Theta}$ be functions in $[[X \rightharpoonup V] \rightharpoonup \mathbb{N}_{\geq 0}]$, where the value of mapping denotes a point in time. The function $\mathcal{T}_{\Delta}$ maps each instance to the time of the initial definition in Δ. This includes definitions of new monitors and derived monitors, as the timestamp is copied in this case. The function $\mathcal{T}_{\Theta}$ returns the last time, an instance was seen, e.g., carried with a base event. Both timestamps can be used to detect the interleaving of e_2 in this case. Providing timestamps, on event $e_2\langle a_1 b_1\rangle$ a check can be performed if there exists an instance less informative than $\langle a_1 b_1\rangle$, which have been seen after the monitor for $\langle a_1\rangle$ has been created.

Monitor creation time can also be used to fix false positives in the second class of problematic traces. Consider above example with e_1 and e_3 annotated as creation events. Table 2.5 shows Δ and Γ for input trace $e_3\langle b_1\rangle e_1\langle a_1\rangle e_2\langle a_1 b_1\rangle$. Since e_3 is a creation event, the trace will be considered for matching on this event. This directly invalidates the pattern specified with $e_1 e_2$ for any trace slice containing the $b \mapsto b_1$ mapping. But as visible in Table 2.5, the definition of the first monitor has no impact on the creation of the $\langle a_1 b_1\rangle$ monitor, and the invalidation stays unrecognized.

Instance timestamps are no help, since $\mathcal{T}_{\Theta}(\langle b_1\rangle) < \mathcal{T}_{\Delta}(\langle a_1\rangle)$. The solution for this problem is to perform checks for existing monitors before the creation of the $\langle a_1 b_1\rangle$ monitor. If there exists a monitor definition for an instance less informative then $\langle a_1 b_1\rangle$ which is not $\langle a_1\rangle$ and created before $\Delta(\langle a_1\rangle)$, then this definition invalidates the trace for the trace slice containing it. The reason for this can be explained in conjunction with enable sets. Assume, e_3 would be a

$e\langle\theta\rangle$	Δ	Γ
$e_3\langle b_1\rangle$	$\langle b_1\rangle \mapsto \sigma(s_0, e_3)$	$\langle b_1\rangle \mapsto$ fail
$e_1\langle a_1\rangle$	$\langle a_1\rangle \mapsto \sigma(s_0, e_1)$	$\langle a_1\rangle \mapsto$?
	$\langle b_1\rangle \mapsto \sigma(s_0, e_3)$	$\langle b_1\rangle \mapsto$ fail
$e_2\langle a_1 b_1\rangle$	$\langle a_1\rangle \mapsto \sigma(s_0, e_1)$	$\langle a_1\rangle \mapsto$?
	$\langle b_1\rangle \mapsto \sigma(s_0, e_3)$	$\langle b_1\rangle \mapsto$ fail
	$\langle a_1 b_1\rangle \mapsto \sigma(\sigma(s_0, e_1), e_2)$	$\langle a_1 b_1\rangle \mapsto$ match

Table 2.5.: Algorithm $\mathbb{C}^+\langle X\rangle$ considering enable sets and trace $e_3\langle b_1\rangle, e_1\langle a_1\rangle, e_2\langle a_1 b_1\rangle$ (e_1, e_2 are creation events)

valid start of a valid trace, then its parameters would be in the enable set of e_1, leading to the creation of a $\langle a_1 b_1\rangle$ monitor derived from $\Delta(b_1)$. Because this monitor does not exist at the time $e_2\langle a_1 b_1\rangle$ is encountered, this implies that e_3 is not a valid start of the trace and the associated instance is invalidated for this trace slice.

Algorithm $\mathbb{D}\langle X\rangle$ [14] formulates a sound solution how to perform online monitoring using timestamps. The generalization of the described rules is specified as a check when creating monitor definitions for an instance θ derived from an instance θ'. Before the definition, algorithm $\mathbb{D}\langle X\rangle$ performs a time check for all instances $\theta'' \in [X \rightharpoonup V]$ with $\theta'' \sqsubseteq \theta$ and $\theta'' \not\sqsubseteq \theta'$. The time check fails if $\mathcal{T}_\Theta(\theta'') > \mathcal{T}_\Delta(\theta')$ or $\mathcal{T}_\Delta(\theta'') < \mathcal{T}_\Delta(\theta')$ is true.

Note that the check for existing monitors in algorithm $\mathbb{D}\langle X\rangle$, Line 26-30, is a simplified version of the time check for the class of cases sketched with above example. Due to the definition of the enable sets, an instance less informative than the input instance may not be defined in Δ. The exact time of creation of the existing monitor is irrelevant in this case.

$\textsc{CalcEnableSets}_{\mathsf{fsm}}(P)$

```
Input:   $\textsc{Reachable}_{\mathcal{G}} \subseteq S$          ▷ states on path to accepting states for $\mathcal{G}$
Globals: $\textsc{Seen} : S \rightarrow \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$
Output:  $\textsc{Enable}^{\mathcal{E}}_{\mathcal{G}} : \mathcal{E} \rightarrow \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$
         $\textsc{Enable}^{X}_{\mathcal{G}} : \mathcal{E} \rightarrow \mathcal{P}_f(\mathcal{P}_f(X))$
Init:    $\textsc{Seen}(s) \leftarrow \emptyset$ for all $s \in S$
         $\textsc{Enable}^{\mathcal{E}}_{\mathcal{G}}(e) \leftarrow \emptyset$, $\textsc{Enable}^{X}_{\mathcal{G}}(e) \leftarrow \emptyset$ for all $e \in \mathcal{E}$

 1: function $\textsc{Main}$
 2:     $\textsc{CalcEnableEventSets}(s_0, \emptyset)$
 3:     $\textsc{CalcEnableParameterSets}()$
 4: end function

 5: function $\textsc{CalcEnableEventSets}(s, E)$
 6:     $\textsc{Seen}(s) \leftarrow \textsc{Seen}(s) \cup \{E\}$
 7:     for all $e \in \mathcal{E}$ do
 8:         $s' \leftarrow \sigma(s, e)$
 9:         if $s' \in \textsc{Reachable}_{\mathcal{G}}$ then
10:             $\textsc{Enable}^{\mathcal{E}}_{\mathcal{G}}(e) \leftarrow \textsc{Enable}^{\mathcal{E}}_{\mathcal{G}}(e) \cup \{E \setminus \{e\}\}$
11:             if $E \cup e \notin \textsc{Seen}(s')$ then
12:                 $\textsc{CalcEnableEventSets}(s', E \cup e)$
13:             end if
14:         end if
15:     end for
16: end function

17: function $\textsc{CalcEnableParameterSets}()$
18:     for all $e \in \mathcal{E}$ do
19:         for all $E \in \textsc{Enable}^{\mathcal{E}}_{\mathcal{G}}(e)$ do
20:             $\mathcal{X} \leftarrow \emptyset$
21:             for all $e' \in E$ do
22:                 $\mathcal{X} \leftarrow \mathcal{X} \cup \mathcal{D}_{\mathcal{E}}(e')$
23:             end for
24:             $\textsc{Enable}^{X}_{\mathcal{G}}(e) \leftarrow \textsc{Enable}^{X}_{\mathcal{G}}(e) \cup \mathcal{X}$
25:         end for
26:     end for
27: end function
```

Based on Chen et al.[14]

Algorithm $\mathbb{D}\langle X\rangle(M = (S, \mathcal{E}, \mathcal{C}, s_0, \sigma, \gamma))$

```
Input:    $\textsc{Enable} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(X))$
Globals:  $\Delta : [X \rightharpoonup V] \rightharpoonup S$
          $\mathcal{U} : [X \rightharpoonup V] \to \mathcal{P}_f([X \rightharpoonup V])$
          $\mathcal{T}_\Delta, \mathcal{T}_\Theta : [X \rightharpoonup V] \rightharpoonup \mathbb{N}$
          $t \in \mathbb{N}$
Init:     $\mathcal{U}(\theta) \leftarrow \emptyset$ for any $\theta \in [X \rightharpoonup V]$; $t \leftarrow 0$

 1: function Main($e\langle\theta\rangle$)
 2:     if $\Delta(\theta)$ undefined then
 3:         CreateNewMonitorStates($e\langle\theta\rangle$)
 4:         if $\Delta(\theta)$ undefined and $e$ is creation event then
 5:             DefineNew($\theta$)
 6:         end if
 7:         $\mathcal{T}_\Theta(\theta) \leftarrow t$; $t \leftarrow t + 1$
 8:     end if
 9:     for all $(\theta') \in \{\theta\} \cup \mathcal{U}(\theta)$ s.t. $\Delta(\theta')$ defined do
10:         $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$
11:     end for
12: end function

13: function CreateNewMonitorStates($e\langle\theta\rangle$)
14:     for all $\mathcal{X}_e \in \textsc{Enable}(e)$ do
15:         if $\textsc{Dom}(\theta) \nsubseteq \mathcal{X}_e$ then
16:             $\theta_m \leftarrow \theta'$ s.t. $\theta' \sqsubset \theta$ and $\textsc{Dom}(\theta') = \textsc{Dom}(\theta) \cap \mathcal{X}_e$
17:             for all $\theta'' \in \mathcal{U}(\theta_m) \cup \{\theta_m\}$ s.t. $\textsc{Dom}(\theta'') = \mathcal{X}_e$ do
18:                 if $\Delta(\theta'')$ defined and $\Delta(\theta'' \sqcup \theta)$ undefined then
19:                     DefineTo($\theta'' \sqcup \theta, \theta''$)
20:                 end if
21:             end for
22:         end if
23:     end for
24: end function

25: function DefineNew($\theta$)
26:     for all $\theta' \sqsubset \theta$ do
27:         if $\Delta(\theta')$ defined then
28:             return
29:         end if
30:     end for
31:     $\Delta(\theta) \leftarrow s_0$; $\mathcal{T}_\Delta(\theta) \leftarrow t$; $t \leftarrow t + 1$
32:     for all $\theta' \sqsubset \theta$ do
33:         $\mathcal{U}(\theta') \leftarrow \mathcal{U}(\theta') \cup \{\theta\}$
34:     end for
35: end function

36: function DefineTo($\theta, \theta'$)
37:     for all $\theta'' \sqsubseteq \theta$ s.t. $\theta'' \nsqsubseteq \theta'$ do
38:         if $\mathcal{T}_\Theta(\theta'') > \mathcal{T}_\Delta(\theta')$ or $\mathcal{T}_\Delta(\theta'') < \mathcal{T}_\Delta(\theta')$ then
39:             return
40:         end if
41:     end for
42:     $\Delta(\theta) \leftarrow \Delta(\theta')$; $\mathcal{T}_\Delta(\theta) \leftarrow \mathcal{T}_\Delta(\theta')$
43:     for all $\theta'' \sqsubset \theta$ do
44:         $\mathcal{U}(\theta'') \leftarrow \mathcal{U}(\theta'') \cup \{\theta\}$
45:     end for
46: end function
```

Reproduced from Chen et al.[14]

3 Approach

In this chapter, we present our approach for online monitoring of parametric traces. It is based on existing work of Chen and Roşu [11] described in Section 2.2. We present algorithm $\mathbb{D}'\langle X\rangle$, a variant of algorithm $\mathbb{D}\langle X\rangle$, aiming to be suitable for library-based implementations on conventional systems offering array data structures.

3.1 Monitoring Algorithm

We first describe the semantics of our algorithm in comparison to the semantics of algorithm $\mathbb{D}\langle X\rangle$. Then we motivate and describe algorithm $\mathbb{D}'\langle X\rangle$. In Section 3.1.3 we introduce a simplified data structure for algorithm $\mathbb{D}'\langle X\rangle$ and use it to discuss further improvements and aspects of the algorithm and its underlying data structure.

3.1.1 Semantics

In the following two sections we specify when a trace is matched by our algorithm.

3.1.1.1 Total Matching and Partial Matching

As defined in Definition 2, non-parametric properties are functions, mapping words in $\mathcal{E}^*$ to a set of categories. Properties are defined using patterns in logic formalisms. As motivated in Section 2.1.6, we will focus on finite-logic patterns like FSMs or regular expressions to specify regular languages containing all words matching a specific pattern. In Section 2.2.4 we have presented online parametric monitoring with creation events. Creation events allow users to ignore the events in the trace until a specific event, the creation event, is encountered. With creation events, regular patterns as ab, can be configured to match traces like bab and $bbbab$ by annotating the event a as creation event. However, user annotated creation events have to be considered *external* to the pattern defined in the specification formalisms.

As recognizing traces as words in languages targets the entire trace, the addition of external creation events, on the other hand, implies matching a single suffix of this trace.

We will formalize these two concepts of matching as functions over non-parametric properties. As defined in Definition 10, non-parametric properties constitute parametric properties which are a basis for parametric monitors. These properties match the entire trace and, thus, have *total matching semantics*. We define this notion as a function over properties.

Definition 17. ***(Total matching semantics)*** *A property with total matching semantics based on $P \in [\mathcal{E}^* \to \mathcal{C}]$ is defined by the function $[\![\cdot]\!]_{total} : [\mathcal{E}^* \to \mathcal{C}] \to [\mathcal{E}^* \to \mathcal{C}]$ with $[\![P]\!]_{total}(\tau) = P(\tau)$.*

Total matching semantics implies that on each processed event, the whole trace, from the first event to the current event, is matched by each pattern associated with the property's categories. *Example:* Given $\mathcal{E} = \{a, b\}$ and a pattern specified by the regular expression ab, this pattern will only match a single trace, namely ab. Traces like aab or bab will not be matched, just as future occurrences of the pattern. To detect each occurrence of ab, the pattern has to be specified more intricately with $((a \mid b)^*ab)^+$.

Properties with *partial matching semantics* are properties configured with starting events, enabling a total match of a suffix of the trace.

Definition 18. ***(Partial matching semantics)*** *A property with partial matching semantics based on $P \in [\mathcal{E}^* \to \mathcal{C}]$ and a set of starting events $\mathcal{E}_\alpha \subseteq \mathcal{E}$ is defined by the function $[\![\cdot]\!]_{partial} : [\mathcal{E}^* \to \mathcal{C}] \to [\mathcal{P}_f(\mathcal{E}) \to [\mathcal{E}^* \to \mathcal{C}]]$ with*

$$[\![P]\!]_{partial}(\mathcal{E}_\alpha)(\tau) = \begin{cases} P(e\tau_2) & \textit{if } \tau_1 \in (\mathcal{E}\backslash\mathcal{E}_\alpha)^*, \tau_2 \in \mathcal{E}^* \textit{ exist, such that } \tau = \tau_1 e \tau_2 \textit{ and } e \in \mathcal{E}_\alpha \\ P(\epsilon) & \textit{otherwise} \end{cases}$$

Partial matching is total matching enabled by starting events. With partial matching the trace is ignored until an event in $\mathcal{E}_\alpha$ is observed. On the first input event in $\mathcal{E}_\alpha$, the suffix starting from this event will be matched with total matching semantics. Considering the pattern ab and $\mathcal{E}_\alpha = \{a\}$, the traces ab, bab, $bbab$ are matched, since the initial b's are ignored. To implement detection of all occurrences of the pattern, we still have to recur to $((a \mid b)^*ab)^+$. In above example with given pattern ab and $\mathcal{E}_\alpha = \{a\}$, to prevent matching of the traces ab, bab, $bbab$, the event b has to be added to $\mathcal{E}_\alpha$. Note that partial matching with $\mathcal{E}_\alpha = \mathcal{E}$ is total matching, since the relevant suffix of the trace will start from any initial input event.

3.1.1.2 Establishing Total Matching Semantics

Creation events which were introduced as an internal concept in algorithm $\mathbb{C}\langle X\rangle$ are a relevant performance optimization as they prevent the creation of monitors in unnecessary cases. Providing creation events as annotations external to patterns specified in logic formalisms exposes this internal concept and introduces partial matching semantics as defined in Definition 18, with $\mathcal{E}_\alpha$ being the set of creation events. We admit that there may be occasions, where annotation of creation events may appear like a less verbose notation of the specification. There may also exist formalisms, where the creation events may add to the richness of the formalism. However, considering finite-logic properties, we regard this external annotation as avoidable for two reasons.

Firstly, it does not add to the richness of the formalism. It can be shown that the original intention of creation events can be fully expressed in the FSM without any special differentiation of events. We argue that the set of creation events, as needed by monitoring algorithms $\mathbb{C}\langle X\rangle$ and beyond, can be derived from the FSM.

Secondly, the semantics are considered to be more complicated. Therefore, reasoning about traces may not be improved, but rather impaired. With total matching semantics, finding patterns specified by regular expressions or FSMs means simply solving the word problem for regular languages. Increasing complexity by introducing additions to the formalism, therefore, should have a well-argued cause.

Because of slight doubts about the positive effects at the user level, we consider the semantic consequences as to strong and opt for total matching semantics. Therefore, a way to establish total matching semantics in a variant of algorithm $\mathbb{D}\langle X\rangle$ is proposed. The changes are compatible with algorithm $\mathbb{D}\langle X\rangle$, as they do not affect the online monitoring algorithm itself, but only its auxiliary algorithms which extract information from the property.

Creation events are characterized by Chen et al. as "sets of events which start the monitoring itself". Events which are not creation events are ignored up to this starting point. In the FSM, we can express non-creation events by specifying loops on the initial state for all events which have to be ignored. All other transitions are considered to be labeled by creation events.

Definition 19. ***(Creation events in properties specified by FSMs)*** *Let $P : \mathcal{E}^* \to \mathcal{C}$ be a property specified by the deterministic finite automaton $\phi = (S, \mathcal{E}, s_0, \sigma, A)$, with S the set of states, $\mathcal{E}$ the set of symbols, s_0 the initial state, $\sigma : S \times \mathcal{E} \to \mathcal{E}$ the transition function, $A \subseteq S$ the set of accepting states. The set of creation events for a property P specified by ϕ is defined as $\textsc{Create}_P^{\mathcal{E}} = \{e \mid e \in \mathcal{E} \wedge \sigma(s_0, e) \neq s_0\}$.*

Events which label transitions to non-initial states are not considered to be creation events, as they affect the monitoring result. Note that transitions to the dead state are creation events, because otherwise pattern like "*a* with no *b* before" could not be specified, as all *b* events would be ignored.

This definition of creation events, however, does not work with the existing definition of enable sets, as introduced in Definitions 14, 15 and 16. Let the pattern ϕ_{umi^*} be specified by the FSM in Figure 3.1 as a variant of the UnsafeMapIterator pattern, with initial self-loops for useIter, createIter and updateMap.

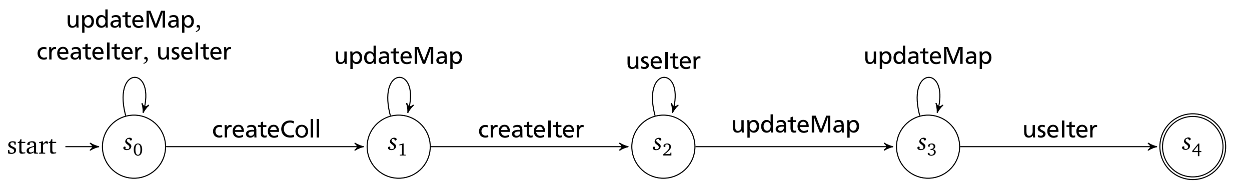


Figure 3.1.: Finite-state machine representing the UnsafeMapIterator pattern with initial self-loops

Let the property $P_{\mathsf{umi}^*} : \mathcal{E}^* \to \mathcal{C}$ be defined as follows:

$$P_{\mathsf{umi}^*}(\tau) = \begin{cases} \mathsf{violation} & \textit{when } \tau \in L(\phi_{\mathsf{umi}^*}) \\ \mathsf{fail} & \textit{when no } \tau' \textit{ exists in } \mathcal{E}^* \textit{ such that } \tau\tau' \in L(\phi_{\mathsf{umi}^*}) \\ \mathsf{?} & \textit{otherwise} \end{cases}$$

This specification may seem contrived at first glance, since createIter and useIter preceding createColl appear impossible in practice. Considering the fact that a map can not create a collection which already exists, this observation regarding the successor createColl event is true. However, it is important to notice that createIter and useIter may happen *without* any createColl being encountered in the future belonging to the same trace slice. Not all collections in the program are derived from maps. Collections like lists and sets may create iterators which would count as createIter, while any usage of such iterators would be registered as useIter. This has the consequence that without appropriate extraction of creation events from the initial self-loops the algorithm would create initially dead monitors for these “independent” createIter and useIter events.

Given property ϕ_{umi^*}, we calculate the property-enable parameter sets $\textsc{Enable}^{\mathcal{E}}_{\mathcal{G}}$ with algorithm $\textsc{CalcEnableSets}_{\text{fsm}}$ specified in Section 15. The results are shown in Table 3.1. As notable by comparing with Table 2.3, the enable sets are considerably larger as calculated for property ϕ_{umi} specified in Figure 3.1. The abundance of enable sets is an outcome of the additional self-loops on the initial state, as the loops enable a multitude of additional valid paths to accepting states associated with violation.

e	$\textsc{Enable}^{X}_{\mathcal{G}}$
createColl	$\{\emptyset, \{m\}, \{i\}, \{m,i\}, \{c,i\}, \{m,c,i\}\}$
createIter	$\{\emptyset, \{m\}, \{i\}, \{m,c\}, \{m,i\}, \{m,c,i\}\}$
useIter	$\{\emptyset, \{m\}, \{c,i\}, \{m,c,i\}\}$
updateMap	$\{\emptyset, \{i\}, \{m,c\}, \{c,i\}, \{m,c,i\}\}$

Table 3.1.: $\textsc{Enable}^{X}_{\mathcal{G}}$ for P_{umi^*} with $\mathcal{G} = \{\text{violation}\}$.

It shows that there is an incompatibility with the usage of this creation event definition and the classic definition of enable sets. Monitoring property P_{umi^*} with algorithm $\mathbb{D}\langle X\rangle$ can have negative effects on performance, as compared to P_{umi} with createColl annotated as creation event. The larger enable sets have an impact on the length of the main loop in $\textsc{CreateNewMonitorStates}$ in algorithm $\mathbb{D}\langle X\rangle$. Note that the main intention of the enable set optimization is realized in this case, preventing the creation of combinations of i-instances with existing mc-instances. $\{m,c\} \notin \textsc{Enable}^{X}_{\mathcal{G}}$, therefore, no mci-monitor is created on incoming event useIter$\langle\theta\rangle$.

The reason for the incompatibility is the total matching semantics, which is *embedded* in the trace enable sets defined in Definition 14. Given trace $\tau \in \mathcal{E}^*$ and event $e \in \mathcal{E}$ the trace-enable event set $\textsc{Enable}(\tau, e)$ is calculated over the whole trace. With initial self-loops, this has the consequence that each such loop adds unlimited possible prefixes to the trace without any effect for reaching a verdict category. Enable sets of an event are intended to model all preceding events, which are necessary to reach an accepting state. As these events do not have an impact for reaching verdict categories, this outcome has to be considered a degradation. We, therefore, decide to drop the calculation of enable sets over the whole trace, and propose to derive them from a suffix of the trace, starting with the first creation event. This solution can be considered as establishing partial matching semantics for trace enable sets. We, therefore, propose a variant of trace enable sets depending on creation events, which takes only trace suffixes into account that start on the first creation event.

Definition 20. ***(Partial trace enable set)*** *Given trace $\tau \in \mathcal{E}^*$, let τ_α be the suffix of τ starting on the first occurrence of an event $e_\alpha \in \textsc{Create}^{\mathcal{E}}_{P}$. Given events $e, e' \in \mathcal{E}$, we denote that e' occurs before the first occurrence of e in τ_α as $e' \overset{\circ}{\rightsquigarrow}_\tau e$. Given a property $P : \mathcal{E}^* \to \mathcal{C}$, let the partial trace enable set of $e \in \mathcal{E}$ be defined by the function $\textsc{PartEnable}_P : (\mathcal{E}^* \times \mathcal{E}) \to \mathcal{P}_f(\mathcal{E})$, defined as $\textsc{PartEnable}_P(\tau, e) = \{e' \mid e' \in \tau \wedge e' \overset{\circ}{\rightsquigarrow}_\tau e\}$.*

Because of their dependencies, definitions of adapted property enable event sets and property enable parameter set representations are also necessary. To reach a verdict category, a *partial property-enable event set* comprises all events which have to occur before the first occurrence of e after a creation event was encountered.

Definition 21. ***(Partial event enable set)*** *Given $P : \mathcal{E}^* \to \mathcal{C}$ and a set of categories specified as goal $\mathcal{G} \subseteq \mathcal{C}$, the partial event enable set is defined by the function $\textsc{PartEnable}^{\mathcal{E}}_{\mathcal{G}} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$ with $\textsc{PartEnable}^{\mathcal{E}}_{\mathcal{G}}(e) = \{\textsc{PartEnable}_P(\tau, e) \mid \tau \in \mathcal{E}^* \wedge P(\tau) \in \mathcal{G}\}$.*

Definition 22. ***(Partial parameter enable set)*** *Given a property $P : \mathcal{E}^* \to \mathcal{C}$, a set of verdict categories $\mathcal{G} \subseteq \mathcal{C}$, a set of parameters X, the partial parameter enable set of event $e \in \mathcal{E}$ is defined by the function $\textsc{PartEnable}^{X}_{\mathcal{G}} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(X))$ with $\textsc{PartEnable}^{X}_{\mathcal{G}}(e) = \{\cup\{\mathcal{D}_{\mathcal{E}}(e') \mid e' \in \textsc{PartEnable}_P(\tau, e)\} \mid \tau \in \mathcal{E}^* \wedge P(\tau) \in \mathcal{G}\}$.*

An algorithm to calculate $\textsc{PartEnable}^{\mathcal{E}}_{\mathcal{G}}$ and $\textsc{PartEnable}^{X}_{\mathcal{G}}$ can be formulated as a variant of algorithm $\textsc{CalcEnableSets}_{\text{fsm}}$. We only have to restrict the traversal through the FSM by treating initial self-looping transitions as if they would point to the dead state. Given property P_{umi^*}, algorithm $\textsc{CalcPartEnableSets}_{\text{fsm}}$ calculates $\textsc{PartEnable}^{\mathcal{E}}_{\mathcal{G}}$ and $\textsc{PartEnable}^{X}_{\mathcal{G}}$ as depicted in Table 3.2.

As can be derived by comparison with Table 2.3, the results are identical to calculating $\textsc{Enable}^{\mathcal{E}}_{\mathcal{G}}$ and $\textsc{Enable}^{X}_{\mathcal{G}}$ using $\textsc{CalcEnableSets}_{\text{fsm}}$ and property P_{umi}. It can be concluded that although having subtle semantical differences, both specifications paired with their appropriate enable set definitions show equivalent behavior in the algorithm.

e	$\textsc{PartEnable}_{\mathcal{G}}^{\mathcal{E}}$	$\textsc{PartEnable}_{\mathcal{G}}^{X}$
createColl	$\{\emptyset\}$	$\{\emptyset\}$
createIter	$\{\{\text{createColl}\}, \{\text{createColl}, \text{updateMap}\}\}$	$\{\{m, c\}\}$
useIter	$\{\{\text{createColl}, \text{createIter}\}, \{\text{createColl}, \text{updateMap}, \text{createIter}\}\}$	$\{\{m, c, i\}\}$
updateMap	$\{\{\text{createColl}\}, \{\text{createColl}, \text{createIter}\}, \{\text{createColl}, \text{updateMap}, \text{createIter}\}\}$	$\{\{m, c\}, \{m, c, i\}\}$

Table 3.2.: $\textsc{PartEnable}_{\mathcal{G}}^{\mathcal{E}}$ and $\textsc{PartEnable}_{\mathcal{G}}^{X}$ for P_{umi^*} with $\mathcal{G} = \{\text{violation}\}$.

```
CalcPartEnableSets_fsm(P)

Input:    $\textsc{Reachable}_{\mathcal{G}} \subseteq S$          ▷ states on path to accepting states for $\mathcal{G}$
Globals:  $\textsc{Seen} : S \to \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$
Output:   $\textsc{PartEnable}_{\mathcal{G}}^{\mathcal{E}} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$
          $\textsc{PartEnable}_{\mathcal{G}}^{X} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(X))$
Init:     $\textsc{Seen}(s) \leftarrow \emptyset$ for all $s \in S$
          $\textsc{PartEnable}_{\mathcal{G}}^{\mathcal{E}}(e) \leftarrow \emptyset$, $\textsc{PartEnable}_{\mathcal{G}}^{X}(e) \leftarrow \emptyset$ for all $e \in \mathcal{E}$

function Main
    CalcPartEnableEventSets($s_0, \emptyset$)
    CalculatePartEnableParameterSets()
end function

function CalcPartEnableEventSets($s, E$)
    $\textsc{Seen}(s) \leftarrow \textsc{Seen}(s) \cup \{E\}$
    for all $e \in \mathcal{E}$ do
        $s' \leftarrow \sigma(s, e)$
        if $s' \in \textsc{Reachable}_{\mathcal{G}} \wedge \neg(s = s_0 \wedge s = s')$ then
            $\textsc{PartEnable}_{\mathcal{G}}^{\mathcal{E}}(e) \leftarrow \textsc{PartEnable}_{\mathcal{G}}^{\mathcal{E}}(e) \cup \{E \backslash \{e\}\}$
            if $E \cup e \notin \textsc{Seen}(s')$ then
                CalcPartEnableEventSets($s', E \cup e$)
            end if
        end if
    end for
end function

function CalcPartEnableParameterSets()
    for all $e \in \mathcal{E}$ do
        for all $E \in \textsc{PartEnable}_{\mathcal{G}}^{\mathcal{E}}(e)$ do
            $\mathcal{X} \leftarrow \emptyset$
            for all $e' \in E$ do
                $\mathcal{X} \leftarrow \mathcal{X} \cup \mathcal{D}_{\mathcal{E}}(e')$
            end for
            $\textsc{PartEnable}_{\mathcal{G}}^{X}(e) \leftarrow \textsc{PartEnable}_{\mathcal{G}}^{X}(e) \cup \mathcal{X}$
        end for
    end for
end function
```

3.1.2 Algorithm $\mathbb{D}'\langle X \rangle$

Each iteration of algorithm $\mathbb{D}\langle X \rangle$ performed on an input event implies various computations based on the event and mutations of data structures necessary for maintaining a monitor state for each trace slice. It can be shown that many computations do not depend on the parameter instance carried by event $e\langle \theta \rangle$, but on its domain $\textsc{Dom}(\theta)$, as introduced in Definition 4. Although the space of possible instances is unlimited due to a potentially unbounded number of parameter values, the space of instance domains is finite. It is restricted by the size of the parameter set X, which is known at initialization time, so that $\textsc{Dom}(\theta) \in \{\mathcal{P}_f(X)\}$ holds for all θ in $[X \rightharpoonup V]$. Note that X, as introduced in Definition 3, is always finite in practice. For convenience, we will sometimes refer to $\textsc{Dom}(\theta)$ as the *instance type* of the concrete instance θ.

In this section we will introduce algorithm $\mathbb{D}'\langle X\rangle$, an adaptation of algorithm $\mathbb{D}\langle X\rangle$ described in Section 2.2.4. The former is functionally equivalent to the latter, as it maintains mappings to monitor states of all non-parametric traces, sliced from a potentially infinite stream of parametric events. All functional requirements are, therefore, the same. The algorithm contains all optimizations added since algorithm $\mathbb{C}\langle X\rangle$, e.g., creation events and enable sets. Some additional modifications have been performed, serving with the following contributions:

1. The algorithm is restructured to identify expressions which allow to precompute values based on information which is available at initialization time. These precomputed values can then be passed to expressions depending on information only available at runtime. This enables runtime-efficient implementations performing a number of computations at startup, omitting redundant operations at runtime.

2. Relevant sets, over which operations are performed, are minimized. In conjunction with 1., we can exploit certain information at runtime to decrease the number of function executions. This involves the operations defined in the original functions $\textsc{DefineNew}$ and $\textsc{DefineTo}$, which formalize instructions to create and update monitors. Here we can show that it is sufficient to perform these operations based on a subset of instances carried with base events, as opposed to operations on subsets of all defined instances.

3. A solution for runtime-efficient retrieval of compatible instances, performed in the original $\textsc{DefineNew}$, is formulated. Instead of performing an excessive runtime-filtering over potentially large sets of instances, the algorithm maintains additional sets of instance references, representing only instances of specified interest. Although we are not aware of a formalization, an implementation similar to this solution is used in the current version of JavaMOP. In Section 3.1.3.1, we improve this approach, allowing to decrease its memory footprint, and create the foundation for the data structure described in Section 3.2.

The restructuring is performed by applying a substitution of computations, depending on concrete instances, by computations, depending on their types. Recall that the type of instances is encoded in the parameter set associated with the base event, i.e., $\textsc{Dom}(\theta) = \mathcal{D}_{\mathcal{E}}(e)$, for each instance carried by an event $e\langle\theta\rangle \in \mathcal{E} \times [X \rightharpoonup V]$. Most computations target instances with certain relations to other instances, i.e., the $\sqsubset$-relation introduced in Definition 7, or similar. Because in the online monitoring context, operations have to be performed on concrete instances, there has to be a point where information derived from computations on parameters can be applied to instances.

We, therefore, introduce *sub-instances*.

Definition 23. ***(Sub-instances)*** *Given an instance $\theta \in [X \rightharpoonup V]$ and a set of parameters $\mathcal{X} \subset X$, let $\cdot \upharpoonright \cdot$ be the partial function in $[([X \rightharpoonup V] \times \mathcal{P}_f(X)) \rightharpoonup [X \rightharpoonup V]]$ defined as*

$$\theta \upharpoonright \mathcal{X} = \begin{cases} \theta_\lambda(x) = \theta(x) \textit{ if } x \in \mathcal{X}, \textit{ else undefined} & \textit{when } \mathcal{X} \subseteq \textsc{Dom}(\theta) \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

*The instance $\theta \upharpoonright \mathcal{X}$ is also called **sub-instance** of θ,* masked *by parameter set $\mathcal{X}$.*

Example: Let $X = \{x_1, x_2, x_3\}$, $V = \{v_1, v_2, v_3\}$ and $\theta = \langle x_1 \mapsto v_1, x_2 \mapsto v_2\rangle$, then $\theta \upharpoonright \{x_2\} = \langle x_2 \mapsto v_2\rangle$, $\theta \upharpoonright \emptyset = \bot$ and $\theta \upharpoonright \{x_1, x_3\}$ is undefined. Note that $\textsc{Dom}(\theta \upharpoonright \mathcal{X}) = \mathcal{X}$ for all $\theta \in [X \rightharpoonup V]$ and $\mathcal{X} \in \mathcal{P}_f(X)$.

Using sub-instances, common expressions involving concrete instances can be substituted using, e.g., $\{\theta' \mid \theta' \sqsubset \theta\} = \{\theta \upharpoonright \mathcal{X} \mid \mathcal{X} \in \mathcal{P}_f(X) \wedge \mathcal{X} \subset \textsc{Dom}(\theta)\}$. If the instance θ is known to be carried by an event $e\langle\theta\rangle \in \mathcal{E} \times [X \rightharpoonup V]$, then $\{\theta' \mid \theta' \sqsubset \theta\} = \{\theta \upharpoonright \mathcal{X} \mid \mathcal{X} \in \mathcal{P}_f(X) \wedge \mathcal{X} \subset \mathcal{D}_{\mathcal{E}}(e)\}$. Note that $\{\mathcal{X} \mid \mathcal{X} \in \mathcal{P}_f(X) \wedge \mathcal{X} \subset \mathcal{D}_{\mathcal{E}}(e)\}$ is a finite set, because X is finite, and can be computed at initialization time, since e is known at initialization time. This is true for all $e \in \mathcal{E}$. Therefore, a function $\lambda^X_{\sqsubset} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(\mathcal{X}))$ may be precomputed, which returns a set of parameter sets, such that for a given event $e\langle\theta\rangle$, $\{\theta' \mid \theta' \sqsubset \theta\} = \{\theta \upharpoonright \mathcal{X} \mid \mathcal{X} \in \lambda^X_{\sqsubset}(e)\}$.

The pre-computation of such sets, where elements serve as masks for the sub-instance function, is useful because it allows to provide precomputed argument sets for operations needed in real-time. Note that in algorithm $\mathbb{D}'\langle X\rangle$, to avoid needless complexity, $\textsc{Dom}(\theta)$ is preferred over $\mathcal{D}_{\mathcal{E}}(e)$. Remember that the two are identical. Where a computation seemingly depends on a concrete instance, it can be in fact computed beforehand.

3.1.2.1 Algorithm Overview

Algorithm $\mathbb{D}'\langle X\rangle$ requires three inputs at initialization time. $\textsc{Base}^X_P$ is the set of all base event definitions which are defined for a property, i.e., $\textsc{Base}^X_P = \{\mathcal{D}_{\mathcal{E}}(e) \mid e \in \mathcal{E}\}$. The set of creation events $\textsc{Create}^{\mathcal{E}}_P$ is extracted from the property

P as defined in Definition 19. $\textsc{PartEnable}_G^X$ represents the list of partial parameter enable sets for each given event e in reversed topological ordering. As explained in Section 3.1.1.2, it can, in this context, be considered as analogous to the function defined in algorithm $\mathbb{D}\langle X\rangle$.

The globals of algorithm $\mathbb{D}'\langle X\rangle$ are a real superset of the globals of algorithm $\mathbb{D}\langle X\rangle$ comprising function Δ mapping instances to monitor states and the function $\mathcal{U}$ providing sets of more informative instances for a given instance. Timestamps for monitor creation $\mathcal{T}_\Delta$ and last-seen time $\mathcal{T}_\Theta$ are maintained together with the current timestamp t. The function $\mathcal{U}_c$ is introduced, which maintains sets of instances per tuple $(\theta, \mathcal{X})$, comprising all instances θ' which are strictly more informative than θ with $\textsc{Dom}(\theta') = \mathcal{X}$. As described below, this function provides means to perform combinations of compatible instances in the join phase efficiently. $\mathcal{U}$ and $\mathcal{U}_c$ are initially empty for all instances.

The algorithm is divided into five functions:

Function $\textsc{Main}$ can be considered identical to $\textsc{Main}_{\mathbb{D}\langle X\rangle}$, apart from minor changes regarding function signatures. Each incoming undefined instance is used as a basis for the creation of monitors derived from existing ones, and/or to define a new monitor with initial state, if the event is a creation event. At the end of the function, monitor states for instances, more informative than the input instance, are updated.

The function $\textsc{CreateNewMonitorStates}$ is a correctness-preserving transformation of $\textsc{CreateNewMonitorStates}_{\mathbb{D}\langle X\rangle}$. Hence, we can substitute a number of computations based on instances in favor of computations based on events and parameter sets. The original definition compactly generalizes the two phases identified in $\mathbb{C}\langle X\rangle$, named *find-max* and *join* (see Section 2.2.3). We separate these operations again, so that Lines 14-18 represent the find-max phase and Lines 19-25 the join phase.

Given an event $e\langle\theta\rangle$, the find-max phase in algorithm $\mathbb{D}'\langle X\rangle$ is initiated with a search over the list of $\mathcal{X}_m$ ("m" stands for "max"). The search succeeds when a defined sub-instance $\theta \upharpoonright \mathcal{X}_m$ is found, so that θ is defined using the monitor state from $\theta \upharpoonright \mathcal{X}_m$. Recall that the list of $\mathcal{X}_m$ is traversed in topological order derived from the list of enable sets.

In the join phase, an iteration is performed over the list of $\mathcal{X}_j$ ("j" for "join"), which represents the domain of all instances, which can be potentially combined with θ. The function $\mathcal{U}_c$ maps a tuple $(\theta \upharpoonright \mathcal{X}_j \cap \textsc{Dom}(\theta), \mathcal{X}_j)$ to the set of all θ' with $\textsc{Dom}(\theta') = \mathcal{X}_j$, where $\theta \upharpoonright \mathcal{X}_j \cap \textsc{Dom}(\theta)$ denotes the instance containing the bindings, which θ and θ' have in common. This enables the creation of combinations $\theta \sqcup \theta'$ with $\textsc{Dom}(\theta \sqcup \theta') = \mathcal{X}_j \cup \textsc{Dom}(\theta)$. The soundness of the transformation is shown in the proof of Proposition 1.

On each execution of $\textsc{DefineNew}$, a search for defined instances is performed by iterating through the set of $\mathcal{X}_d$, representing sub-instance masks selecting potentially *disabling* instances $\theta \upharpoonright \mathcal{X}_d$. If $\Delta(\theta \upharpoonright \mathcal{X}_d)$ is defined, then the monitor for θ will never reach an accepting state, because although $\theta \upharpoonright \mathcal{X}_d$ is defined before $e\langle\theta\rangle$ is encountered, it is not in one of the enable sets of e. Note that we do not check all sub-instances of θ for a disabling definition, but only the set of base event parameter sets which are subsets of $\textsc{Dom}(\theta)$. The intuition that checking instances of base events is sufficient is based on the following observation: If $\Delta(\theta'')$ is defined and found by the check, with θ'' derived from the instance θ', and θ' was carried by a base event, then θ' is also found. The validity of this observation is shown in the proof of Proposition 3.

The auxiliary function $\textsc{DefineTo}$ accepts the argument list comprising θ and $\mathcal{X}'$ with $\theta \in [X \rightharpoonup V]$, $\mathcal{X}' \in \mathcal{P}_f(X)$. Note that θ is either an instance carried by a base event if called from a find-max phase, or the combination of an existing instance and an input instance if called from the join phase. At the beginning of the method the set of parameter masks $\mathcal{X}_d$ is iterated, selecting disabling sub-instances. An sub-instance of θ disables the creation of the current monitor if the time check defined in Line 38 fails, i.e., evaluates to true. If it evaluates to false for all $\mathcal{X}_d$, a monitor for θ is created, using the monitor state of $\theta \upharpoonright \mathcal{X}$. The time check is introduced in Section 2.2.4, and prevents the creation of monitors in certain contexts. It detects if events belonging to the trace slice for θ have been encountered which invalidate any future match for this trace slice. The creation of a monitor in this is incorrect, as it would allow matches for this trace slice in the future. Note that the time check is only relevant, because the enable set optimization prevents the creation of monitors for certain traces which will never reach a verdict category. In simpler online monitoring algorithms like $\mathbb{C}\langle X\rangle$, monitors are created which capture this information using a dead monitor state. Our definition of the time check deviates from the definition in $\textsc{DefineTo}_{\mathbb{D}\langle X\rangle}$, as it is performed over a constrained list of base event definitions, and not over the constrained set $\mathcal{P}_f(X)$. We can show that checking instances of base events is sufficient, with similar argumentation as in the description of $\textsc{DefineTo}$. The interested reader is referred to Proposition 4 for a proof.

$\textsc{UpdateChainings}$ encapsulates definitions of mappings in the functions $\mathcal{U}$ and $\mathcal{U}_c$. Its input argument is an instance $\theta \in [X \rightharpoonup V]$. Assuming an incoming event $e\langle\theta\rangle$, the $\mathcal{U}$-function is used in $\textsc{Main}$ to update all instance monitors for θ', where $\theta \sqsubset \theta'$, by advancing their monitor states processing base event e. Note that $\mathcal{U}$ is used in $\textsc{Main}$ only in conjunction with instances carried by base events. Therefore, it is sufficient that new mappings in $\mathcal{U}$ are only defined where $\textsc{Dom}(\theta) \in \textsc{Base}_P^X$, since any other mappings will never be retrieved in $\textsc{Main}$. In this point we deviate from the original algorithm, as the for-loop in Line 45 is an iteration over a constrained set of base event parameters, which stands in opposition to the constrained powerset $\mathcal{P}_f(X)$ in $\mathbb{D}\langle X\rangle$. As the proof is merely a reiteration of above reasoning, it is left to the reader.

$\mathcal{U}_c$ maintains mappings similar to $\mathcal{U}$, i.e., $\theta'' \in \mathcal{U}_c(\theta', \mathcal{X})$ with $\theta'' \sqsubset \theta'$ for all $\theta' \in [X \rightharpoonup V]$ and $\mathcal{X} \in \mathcal{P}_f(X)$. In addition, it is required that $\textsc{Dom}(\theta'') = \mathcal{X}$ for all $\theta'' \in \mathcal{U}_c(\theta', \mathcal{X})$. $\mathcal{U}_c$ is useful, because it enables retrieval of all instances θ'' which are compatible with an instance θ_x expected in the future. Definitions of $\mathcal{U}_c$ are performed with the following reasoning: Let θ_c ("c" for "common") and $\mathcal{X}$ be the input arguments for $\mathcal{U}_c$. Furthermore, let θ_x the expected instance which is compatible with all $\theta'' \in \mathcal{U}_c(\theta_c, \mathcal{X})$. In this case, θ_c represents the part of the instance θ_x, which it has in common with all instances θ'' in $\mathcal{U}_c(\theta_c, \mathcal{X})$. Each pair of input arguments θ and $\mathcal{X}$ must, therefore, augment all sets for $\mathcal{U}_c(\theta \upharpoonright \mathcal{X}_c, \mathcal{X})$ with θ, where each $\mathcal{X}_c$ masks the part, which it has in common with an expected instance encountered in the future. Note that the stored θ is the θ_j viewed from the perspective of CreateNewMonitorStates. Since future instances θ_x can only be carried by future events, the X_c are derived from a set based on $\textsc{Base}^X_P$. Proposition 2 ensures that $\mathcal{U}_c$ in fact stores all combination partners for future instances.

```
Algorithm $\mathbb{D}'\langle X\rangle(M = (\mathcal{S}, \mathcal{E}, \mathcal{C}, s_0, \sigma, \gamma))$ (pt.1)

Input:    $\textsc{Base}^X_P \subseteq \mathcal{P}_f(X)$
          $\textsc{Create}^{\mathcal{E}}_P \subseteq \mathcal{E}$
          $\textsc{PartEnable}^X_{\mathcal{G}} : \mathcal{E} \to \{\langle \mathcal{X}_i, ..., \mathcal{X}_k\rangle \mid \mathcal{X}_i, \mathcal{X}_k \in \mathcal{P}_f(X)\}$
Globals:  $\Delta : [X \rightharpoonup V] \rightharpoonup S$
          $\mathcal{U} : [X \rightharpoonup V] \to \mathcal{P}_f([X \rightharpoonup V])$
          $\mathcal{U}_c : ([X \rightharpoonup V] \times \mathcal{P}_f(X)) \to \mathcal{P}_f([X \rightharpoonup V])$
          $\mathcal{T}_\Theta, \mathcal{T}_\Delta : [X \rightharpoonup V] \to \mathbb{N}$
          $t \in \mathbb{N}$
Init:     $\mathcal{U}(\theta) \leftarrow \emptyset$ for any $\theta \in [X \rightharpoonup V]$
          $\mathcal{U}_c(\theta, \mathcal{X}) \leftarrow \emptyset$ for any $\theta \in [X \rightharpoonup V]$ and $\mathcal{X} \in \mathcal{P}_f(X)$
          $\mathcal{T}_\Theta(\theta) = 0$ for any $\theta \in [X \rightharpoonup V]$
          $\mathcal{T}_\Delta(\theta) = \infty_+$ for any $\theta \in [X \rightharpoonup V]$
          $t \leftarrow 0$

 1: function Main($e\langle\theta\rangle$)
 2:     if $\Delta(\theta)$ undefined then
 3:         CreateNewMonitorStates($e\langle\theta\rangle$)
 4:         if $\Delta(\theta)$ undefined and $e \in \textsc{Create}^{\mathcal{E}}_P$ then
 5:             DefineNew($\theta$)
 6:         end if
 7:         $\mathcal{T}_\Theta(\theta) \leftarrow t$; $t \leftarrow t + 1$
 8:     end if
 9:     for all $\theta' \in \{\theta\} \cup \mathcal{U}(\theta)$ s.t. $\Delta(\theta')$ defined do
10:         $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$
11:     end for
12: end function

13: function CreateNewMonitorStates($e\langle\theta\rangle$)
14:     for all $\mathcal{X}_m \in \langle \mathcal{X}_e \mid \mathcal{X}_e \in \textsc{PartEnable}^X_{\mathcal{G}}(e) \wedge X_e \subset \textsc{Dom}(\theta)\rangle$ do
15:         if $\Delta(\theta \upharpoonright \mathcal{X}_m)$ defined and $\Delta(\theta)$ undefined then
16:             DefineTo($\theta, \mathcal{X}_m$)
17:         end if
18:     end for
19:     for all $\mathcal{X}_j \in \langle \mathcal{X}_e \mid \mathcal{X}_e \in \textsc{PartEnable}^X_{\mathcal{G}}(e) \wedge \textsc{Dom}(\theta) \nsubseteq \mathcal{X}_e \wedge \mathcal{X}_e \not\subset \textsc{Dom}(\theta)\rangle$ do
20:         for all $\theta_j \in \mathcal{U}_c(\theta \upharpoonright \mathcal{X}_j \cap \textsc{Dom}(\theta), \mathcal{X}_j)$ do
21:             if $\Delta(\theta_j)$ defined and $\Delta(\theta_j \sqcup \theta)$ undefined then
22:                 DefineTo($\theta_j \sqcup \theta, \mathcal{X}_j$)
23:             end if
24:         end for
25:     end for
26: end function
```

Algorithm $\mathbb{D}'\langle X\rangle(M = (S, \mathcal{E}, \mathcal{C}, s_0, \sigma, \gamma))$ (pt.2)

```
function DefineNew($\theta$)
    for all $\mathcal{X}_d \in \{\mathcal{X}_b \mid \mathcal{X}_b \in \textsc{Base}_P^X \wedge \mathcal{X}_b \subset \textsc{Dom}(\theta)\}$ do
        if $\Delta(\theta \upharpoonright \mathcal{X}_d)$ defined then
            return
        end if
    end for
    $\Delta(\theta) \leftarrow s_0;\ \mathcal{T}(\theta) \leftarrow t;\ t = t + 1$
    UpdateChainings($\theta$)
end function

function DefineTo($\theta, \mathcal{X}'$)
    for all $\mathcal{X}_d \in \{\mathcal{X}_b \mid \mathcal{X}_b \in \textsc{Base}_P^X \wedge \mathcal{X}_b \subseteq \textsc{Dom}(\theta) \wedge \mathcal{X}_b \not\subseteq \mathcal{X}'\}$ do
        if $\mathcal{T}_\Theta(\theta \upharpoonright \mathcal{X}_d) > \mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}')$ or $\mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}_d) < \mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}')$ then
            return
        end if
    end for
    $\Delta(\theta) \leftarrow \Delta(\theta \upharpoonright \mathcal{X}');\ \mathcal{T}(\theta) \leftarrow \mathcal{T}(\theta \upharpoonright \mathcal{X}')$
    UpdateChainings($\theta$)
end function

function UpdateChainings($\theta$)
    for all $\mathcal{X}_u \in \{\mathcal{X}_b \mid \mathcal{X}_b \in \textsc{Base}_P^X \wedge \mathcal{X}_b \subset \textsc{Dom}(\theta)\}$ do
        $\mathcal{U}(\theta \upharpoonright \mathcal{X}_u) \leftarrow \mathcal{U}(\theta \upharpoonright \mathcal{X}_u) \cup \{\theta\}$
    end for
    for all $\mathcal{X}_c \in \{\mathcal{X}_b \cap \textsc{Dom}(\theta) \mid \mathcal{X}_b \in \textsc{Base}_P^X \wedge \mathcal{X}_b \not\subseteq \textsc{Dom}(\theta) \wedge \textsc{Dom}(\theta) \not\subset \mathcal{X}_b\}$ do
        $\mathcal{U}_c(\theta \upharpoonright \mathcal{X}_c, \textsc{Dom}(\theta)) \leftarrow \mathcal{U}_c(\theta \upharpoonright \mathcal{X}_c, \textsc{Dom}(\theta)) \cup \{\theta\}$
    end for
end function
```

3.1.2.2 Proof of Correctness

As already described, algorithm $\mathbb{D}'\langle X\rangle$ is an optimized reformulation of $\mathbb{D}\langle X\rangle$ providing a useful structure for an efficient implementation. Algorithm $\mathbb{D}\langle X\rangle$ is provenly correct and stands as the correctness backbone for algorithm $\mathbb{D}'\langle X\rangle$. We will, therefore, use it to formulate proofs based on similarities and differences of our variant with algorithm $\mathbb{D}\langle X\rangle$. Many reformulations represent swapping operations on instances for operations on parameter sets. Most of them should be intuitively understandable, as the rules underlying the transformations are obvious. The most obvious ones can be, therefore, considered as *natural* substitutions, because the transformation rules can be performed mechanically. As stated in the algorithm description, some expressions have been improved, by using *optimizing* substitutions. The crucial point is that all substitutions have to be correctness preserving. For a given instance $\theta \in [X \rightharpoonup V]$, we consider $\{\theta \upharpoonright \mathcal{X} \mid \mathcal{X} \in \mathcal{P}_f(X) \wedge \mathcal{X} \subset \textsc{Dom}(\theta)\}$ as the natural substitution of $\{\theta' \mid \theta' \in [X \rightharpoonup V] \wedge \theta' \sqsubset \theta\}$, in the sense that the sets are identical and substitution legit in all contexts, i.e., all uses of the set in the algorithm. For a certain set of algorithmic contexts, there may be an optimizing substitution $\{\theta \upharpoonright \mathcal{X} \mid \mathcal{X} \in \mathcal{P}_f(X) \wedge \mathcal{X} \subset \textsc{Dom}(\theta) \wedge C(\mathcal{X})\}$ for $\{\theta' \mid \theta' \in [X \rightharpoonup V] \wedge \theta' \sqsubset \theta\}$, where $C(\mathcal{X})$ is a predicate defined on $\mathcal{P}_f(X)$, such that the *observable result* of using any of the two sets is indistinguishable. This result is to be understood in terms of the function of the program, i.e., its correctness, not computational complexity or other non-functional requirements. Therefore, it is possible that the computational complexity may be beneficially affected by the predicate. Intuitively, two algorithms, which only differ in the definition of this set but compute the same output for the same input can be considered equivalent in respect to the input and output. Since it is open to dispute, if algorithms may be legitimately called equivalent [4] or equivalent without restrictions [36], we, therefore, introduce the notion of *observational equivalence*.

Definition 24. ***(Observational equivalence)*** *Let algorithm A' be a transformation of algorithm A. A and A' are observationally equivalent if given the same input, the observable output of A and A' is indistinguishable. Let f be a function defined in algorithm B. Let f' be a function creating the algorithm variant B' by substituting f for f'. f and f' are observationally equivalent if B and B' are observationally equivalent.*

Note that providing rigid definitions of "algorithm", "transformation" and "substitution" is outside of the scope of this thesis. Especially the latter is problematic since the notion of substitution as it is used here demands an implicit embedding of the function in the algorithm with changes to signature and other details. Therefore, it is by no means a solely syntactical operation. Our intention is to provide an intuitively graspable equivalence relation to be used in our argumentation in Section 3.1.2.2, which captures the notion of an optimization or reorganization of computations used in an algorithm without affecting its correctness.

Proposition 1. *The function* $\textsc{CreateNewMonitorStates}_{\mathbb{D}'\langle X\rangle}$ *is observationally equivalent to* $\textsc{CreateNewMonitorStates}_{\mathbb{D}\langle X\rangle}$.

Proof. We will show that $\textsc{CreateNewMonitorStates}_{\mathbb{D}'\langle X\rangle}$ is a correctness-preserving transformation of the original $\textsc{CreateNewMonitorStates}_{\mathbb{D}\langle X\rangle}$, by proving the soundness of the presented transformation steps individually. Figure 3.2 shows a slight adaptation of the function as it was defined in algorithm $\mathbb{D}\langle X\rangle$, to match the signature of the function $\textsc{DefineTo}_{\mathbb{D}'\langle X\rangle}$.

```
1:  function CreateNewMonitorStates($e\langle\theta\rangle$)
2:      for all $\mathcal{X}_e \in \textsc{PartEnable}^{X}_{\mathcal{G}}(e)$ do
3:          if $\textsc{Dom}(\theta) \not\subseteq \mathcal{X}_e$ then
4:              $\theta_m \leftarrow \theta'$ s.t. $\theta' \sqsubset \theta$ and $\textsc{Dom}(\theta') = \textsc{Dom}(\theta) \cap \mathcal{X}_e$
5:              for all $\theta'' \in \mathcal{U}(\theta_m) \cup \{\theta_m\}$ s.t. $\textsc{Dom}(\theta'') = \mathcal{X}_e$ do
6:                  if $\Delta(\theta'')$ defined and $\Delta(\theta'' \sqcup \theta)$ undefined then
7:                      DefineTo($\theta'' \sqcup \theta, \mathcal{X}_e$)
8:                  end if
9:              end for
10:         end if
11:     end for
12: end function
```

Figure 3.2.: Adapted $\textsc{CreateNewMonitorStates}_{\mathbb{D}\langle X\rangle}$ to match the signature of $\textsc{DefineTo}_{\mathbb{D}'\langle X\rangle}$

Figure 3.3 is the first step in the transformation. The expression in Figure 3.3, Line 3 is defined as a set in terms of

```
1:  function CreateNewMonitorStates($e\langle\theta\rangle$)
2:      for all $\mathcal{X}_e \in \textsc{PartEnable}^{X}_{\mathcal{G}}(e)$ do
3:          for all $\mathcal{X}_m \in \{\mathcal{X}' \mid \mathcal{X}' \in \mathcal{P}_f(X) \wedge \mathcal{X}' = \textsc{Dom}(\theta) \cap \mathcal{X}_e \wedge \textsc{Dom}(\theta) \not\subseteq \mathcal{X}_e\}$ do
4:              for all $\theta'' \in \{\theta' \mid \theta' \in \mathcal{U}(\theta \upharpoonright \mathcal{X}_m) \wedge \textsc{Dom}(\theta') = \mathcal{X}_e\} \cup \{\theta \upharpoonright \mathcal{X}' \mid \mathcal{X}' \in \{\mathcal{X}_m\} \wedge \mathcal{X}_m = \mathcal{X}_e\}$ do
5:                  if $\Delta(\theta'')$ defined and $\Delta(\theta'' \sqcup \theta)$ undefined then
6:                      DefineTo($\theta'' \sqcup \theta, \mathcal{X}_e$)
7:                  end if
8:              end for
9:          end for
10:     end for
11: end function
```

Figure 3.3.: First transformation step of $\textsc{CreateNewMonitorStates}_{\mathbb{D}\langle X\rangle}$

sub-instance masks, comprising all constraints as restrictions to its elements, i.e., $\{\mathcal{X}_m \mid \mathcal{X}' \in \mathcal{P}_f(X) \wedge \mathcal{X}' \subset \textsc{Dom}(\theta) \wedge \mathcal{X}' = \textsc{Dom}(\theta) \cap \mathcal{X}_e \wedge \textsc{Dom}(\theta) \not\subseteq \mathcal{X}_e \wedge \mathcal{X}_e\}$ with $\mathcal{X}_e$ the current item in sequence in $\textsc{PartEnable}^{X}_{\mathcal{G}}(e)$. Note that the set contains either a single element, which is $\mathcal{X}_m$, or no element, when a constraint is violated. The restriction $\mathcal{X}' \subset \textsc{Dom}(\theta)$ is redundant, since $\mathcal{X}'$ is fixed with $\mathcal{X}' = \textsc{Dom}(\theta) \cap \mathcal{X}_e$. Therefore, Figure 3.3, Line 3 shows a simplification. The two expressions specifying a union in Line 5 are substituted by sets where instance θ_m can be substituted by $\theta \upharpoonright \mathcal{X}_m$. Note that $\theta \upharpoonright \mathcal{X}_m$ is always defined, as guaranteed by the constraint $\mathcal{X}_m = \textsc{Dom}(\theta) \cap \mathcal{X}_e$ derived from the main set definition, which implies $\mathcal{X}_m \subset \textsc{Dom}(\theta)$. The next step is depicted in Figure 3.4. In Line 4 the condition $\mathcal{X}_e \subset \textsc{Dom}(\theta)$ is introduced and both cases are applied to the for-loop in Figure 3.3, Lines 3-9. If $\mathcal{X}_e \subset \textsc{Dom}(\theta)$, then with $\mathcal{X}_m = \textsc{Dom}(\theta) \cap \mathcal{X}_e$ follows $X_m = X_e$. With the established semantics of $\mathcal{U}$, there can not be any $\theta'' \in \mathcal{U}(\theta \upharpoonright X_m)$ with $\textsc{Dom}(\theta'') = \mathcal{X}_m$ since this contradicts the requirement $\theta \sqsubset \theta''$. Hence, $\{\theta' \mid \theta' \in \mathcal{U}(\theta \upharpoonright \mathcal{X}_m) \wedge \textsc{Dom}(\theta') = \mathcal{X}_e\}$ is always empty in this case and can be ignored in the union. Consequently, the union has only $\theta \upharpoonright \mathcal{X}_m$ as element and we can remove the for-loop completely. If $\mathcal{X}_e = \mathcal{X}_m$, then $\mathcal{X}_m \subset \textsc{Dom}(\theta)$, therefore, $\theta \upharpoonright \mathcal{X}_m \sqcup \theta$ is nothing more than θ. $\mathcal{X}_m \subset \textsc{Dom}(\theta)$, hence, $\textsc{Dom}(\theta) \cup \mathcal{X}_m = \textsc{Dom}(\theta)$, implying further simplification in Line 6.

In the other case, if $\mathcal{X}_e \not\subset \textsc{Dom}(\theta)$, the set $\{\theta \upharpoonright \mathcal{X}' \mid \mathcal{X}' \in \{\mathcal{X}_m\} \wedge \mathcal{X}_m = \mathcal{X}_e\}$ is always empty and can be removed from the union due to the following reasoning. If $\mathcal{X}_e \not\subset \textsc{Dom}(\theta)$, then $\mathcal{X}_e$ contains at least one element which is not in $\textsc{Dom}(\theta)$. Knowing that $\mathcal{X}_m = \textsc{Dom}(\theta) \cap \mathcal{X}_e$, we infer $\mathcal{X}_m \neq \mathcal{X}_e$ which violates the restriction in above set.

```
function CreateNewMonitorStates($e\langle\theta\rangle$)
    for all $\mathcal{X}_e \in$ PartEnable$^X_\mathcal{G}(e)$ do
        for all $\mathcal{X}_m \in \{\mathcal{X}' \mid \mathcal{X}' \in \mathcal{P}_f(X) \wedge \mathcal{X}' = \text{Dom}(\theta) \cap \mathcal{X}_e \wedge \text{Dom}(\theta) \not\subseteq \mathcal{X}_e\}$ do
            if $X_e \subset \text{Dom}(\theta)$ then
                if $\Delta(\theta \upharpoonright \mathcal{X}_m)$ defined and $\Delta(\theta)$ undefined then
                    DefineTo($\theta, \mathcal{X}_m$)
                    go to 10
                end if
            else
                for all $\theta'' \in \{\theta' \mid \theta' \in \mathcal{U}(\theta \upharpoonright \mathcal{X}_m) \wedge \text{Dom}(\theta') = \mathcal{X}_e\}$ do
                    if $\Delta(\theta'')$ defined and $\Delta(\theta'' \sqcup \theta)$ undefined then
                        DefineTo($\theta'' \sqcup \theta, \mathcal{X}_e$)
                    end if
                end for
            end if
        end for
    end for
end function
```

Figure 3.4.: Second transformation step of CreateNewMonitorStates$_{\mathbb{D}\langle X\rangle}$

In the last transformation step, with results depicted in Line 3.5, two separate for-loops based on PartEnable$^X_\mathcal{G}$ are introduced. Line 2 shows a list comprehension over PartEnable$^X_\mathcal{G}(e)$ with a simplification of the aggregated constraints.

```
function CreateNewMonitorStates($e\langle\theta\rangle$)
    for all $\mathcal{X}_m \in \langle \mathcal{X}_e \mid \mathcal{X}_e \in$ PartEnable$^X_\mathcal{G}(e) \wedge X_e \subset \text{Dom}(\theta)\rangle$ do
        if $\Delta(\theta \upharpoonright \mathcal{X}_m)$ defined and $\Delta(\theta)$ undefined then
            DefineTo($\theta, \mathcal{X}_m$)
        end if
    end for
    for all $\mathcal{X}_j \in \langle \mathcal{X}_e \mid \mathcal{X}_e \in$ PartEnable$^X_\mathcal{G}(e) \wedge \text{Dom}(\theta) \not\subseteq \mathcal{X}_e \wedge \mathcal{X}_e \not\subset$
    $\text{Dom}(\theta)\rangle$ do
        for all $\theta_j \in \mathcal{U}_c(\theta \upharpoonright \mathcal{X}_j \cap \text{Dom}(\theta), \mathcal{X}_j)$ do
            if $\Delta(\theta_j)$ defined and $\Delta(\theta_j \sqcup \theta)$ undefined then
                DefineTo($\theta_j \sqcup \theta, \mathcal{X}_j$)
            end if
        end for
    end for
end function
```

Figure 3.5.: Final result of the transformation of CreateNewMonitorStates$_{\mathbb{D}\langle X\rangle}$

Since $X_e \subset \text{Dom}(\theta)$ is stricter than $\text{Dom}(\theta) \not\subseteq \mathcal{X}_e$, the latter can be dropped. If $X_e \subset \text{Dom}(\theta)$, then $\text{Dom}(\theta) \cap \mathcal{X}_e$ can only be $\mathcal{X}_e$. All occurrences of $\mathcal{X}_m$ are substituted with $\text{Dom}(\theta) \cap \mathcal{X}_m$ implementing the constraint former located in the set definition. Line 7-11 introduce a second list comprehension. Note that $\mathcal{X}_m$ is renamed in this part to $\mathcal{X}_j$. Function $\mathcal{U}$ with the additional restriction to its co-domain, i.e., $\text{Dom}(\theta') = X_e$, is substituted for function $\mathcal{U}_c$. This substitution is valid since $\mathcal{U}_c$ guarantees $\text{Dom}(\theta') = \mathcal{X}$ for all θ' in $\mathcal{U}_c(\theta, \mathcal{X})$, with $\theta \in [X \rightharpoonup V]$ and $\mathcal{X} \in \mathcal{P}_f(X)$, as ensured by the proof of Proposition 2. □

Proposition 2. *Given $\mathcal{U}_c$, the function in algorithm $\mathbb{D}'\langle X\rangle$, the following holds:*

(1) $\text{Dom}(\theta') = \mathcal{X}$, for all $\theta' \in \mathcal{U}_c(\theta, \mathcal{X})$ and any $\theta \in [X \rightharpoonup V]$ and $\mathcal{X} \in \mathcal{P}_f(X)$.

(2) $\theta \sqsubset \theta'$, for all $\theta' \in \mathcal{U}_c(\theta, \mathcal{X})$ and any $\theta \in [X \rightharpoonup V]$ and $\mathcal{X} \in \mathcal{P}_f(X)$.

(3) Given compatible instances $\theta, \theta' \in [X \rightharpoonup V]$ with $\theta \not\sqsubseteq \theta'$ and $\theta' \not\sqsubset \theta$. With Δ being the function in algorithm $\mathbb{D}'\langle X\rangle$, let $e\langle\theta\rangle$ be an incoming input event with $\Delta(\theta)$ undefined. If $\Delta(\theta')$ is already defined and $\text{Dom}(\theta') \in$ PartEnable$^X_\mathcal{G}(e)$, then θ' will be retrieved from $\mathcal{U}_c$ in CreateNewMonitorStates when processing $e\langle\theta\rangle$.

Proof. (1): $\mathcal{U}_c$ is only defined in UPDATECHAININGS. With θ be the input argument in UPDATECHAININGS, the definition has the form $\mathcal{U}_c(\theta', \text{DOM}(\theta)) \leftarrow \mathcal{U}_c(\theta', \text{DOM}(\theta)) \cup \{\theta\}$, with some $\theta' \in [X \rightharpoonup V]$. Hence, the proposition holds.

(2): Let θ be the input argument in UPDATECHAININGS. Then the set $\mathcal{U}_c(\theta \upharpoonright \mathcal{X}_c, \text{DOM}(\theta))$ with $\mathcal{X}_c = \mathcal{X}_b \cap \text{DOM}(\theta)$ and $\mathcal{X}_b \in \text{BASE}_{\mathcal{P}}^X$ is augmented with θ. We know that $\mathcal{X}_b \not\subseteq \text{DOM}(\theta)$ and $\text{DOM}(\theta) \not\subset \mathcal{X}_b$, hence, $\mathcal{X}_b \cap \text{DOM}(\theta) \subset \text{DOM}(\theta)$. Follows $\theta \upharpoonright \mathcal{X}_c \sqsubset \theta$, letting the proposition hold.

(3): Event $e\langle\theta\rangle$ is processed in CREATENEWMONITORSTATES and θ and θ' have the properties described in the proposition. We know that $\text{DOM}(\theta') \in \text{PARTENABLE}_{\mathcal{G}}^X(e)$. With $\theta \not\sqsubseteq \theta'$ and $\theta' \not\sqsubset \theta$ follows $\text{DOM}(\theta) \not\subseteq \text{DOM}(\theta')$ and $\text{DOM}(\theta') \not\subset \text{DOM}(\theta)$. Hence, all constraints in Figure 19 are satisfied and $\mathcal{X}_j = \text{DOM}(\theta')$. Because θ' is already defined, an event $e'\langle\theta'\rangle$ must have occurred before $e\langle\theta\rangle$. We only need to show that $\theta' \in \mathcal{U}_c(\theta \upharpoonright \text{DOM}(\theta') \cap \text{DOM}(\theta), \text{DOM}(\theta'))$. If θ' is defined, then it is either defined in DEFINETO or DEFINENEW. In both cases UPDATECHAININGS(θ') is executed and the set of base events $\text{BASE}_{\mathcal{P}}^X$ is iterated. We know that $\mathcal{D}_{\mathcal{E}}(e) \in \text{BASE}_{\mathcal{P}}^X$, therefore, $\mathcal{X}_b = \text{DOM}(\theta)$ in one iteration in Line 49. With θ_{UC}, the input argument of UPDATECHAININGS, the constraints in Line 49 are written as $\mathcal{X}_b \not\subseteq \text{DOM}(\theta_{\text{UC}})$ and $\text{DOM}(\theta_{\text{UC}}) \not\subset \mathcal{X}_b$. They are satisfied, knowing that $\mathcal{X}_b = \text{DOM}(\theta)$ and $\theta_{\text{UC}} = \theta'$ in combination with the initial assumptions $\text{DOM}(\theta) \not\subseteq \text{DOM}(\theta')$ and $\text{DOM}(\theta') \not\subset \text{DOM}(\theta)$. Hence, $\mathcal{X}_c$ must be $\text{DOM}(\theta) \cap \text{DOM}(\theta')$. Therefore, the set to be augmented in Line 50, i.e., $\mathcal{U}_c(\theta_{\text{UC}} \upharpoonright \mathcal{X}_c, \text{DOM}(\theta_{\text{UC}}))$, becomes $\mathcal{U}_c(\theta' \upharpoonright \text{DOM}(\theta) \cap \text{DOM}(\theta'), \text{DOM}(\theta'))$. We only need to show that $\mathcal{U}_c(\theta' \upharpoonright \text{DOM}(\theta) \cap \text{DOM}(\theta'), \text{DOM}(\theta')) = \mathcal{U}_c(\theta \upharpoonright \text{DOM}(\theta') \cap \text{DOM}(\theta), \text{DOM}(\theta'))$, which is the case, if $\theta' \upharpoonright \text{DOM}(\theta) \cap \text{DOM}(\theta') = \theta \upharpoonright \text{DOM}(\theta') \cap \text{DOM}(\theta)$. Since θ' and θ are compatible, this equality holds. □

DEFINENEW and $\text{DEFINENEW}_{\mathbb{D}\langle X\rangle}$ are sufficiently similar, so we only have to show the correctness of the optimized check for existing monitors, as it deviates from algorithm $\mathbb{D}\langle X\rangle$.

Proposition 3. *Let θ be the input argument of $\text{DEFINENEW}_{\mathbb{D}\langle X\rangle}$, $S_{\mathcal{P}}$ be the set $\{\mathcal{X} \mid \mathcal{X} \in \mathcal{P}_f(X) \wedge \mathcal{X} \subset \text{DOM}(\theta)\}$ and S_B be the set $\{\mathcal{X} \mid \mathcal{X} \in \text{BASE}_{\mathcal{P}}^X \wedge \mathcal{X} \subset \text{DOM}(\theta)\}$ used in the for-loop in algorithm $\mathbb{D}'\langle X\rangle$, Line 28. If* DEFINENEW *with the for-loop over $S_{\mathcal{P}}$ is observationally equivalent to $\text{DEFINENEW}_{\mathbb{D}\langle X\rangle}$, then it is also observationally equivalent to* DEFINENEW *with the for-loop over S_B. We assume,* DEFINENEW *with the for-loop over $S_{\mathcal{P}}$ is observationally equivalent to $\text{DEFINENEW}_{\mathbb{D}\langle X\rangle}$, because $S_{\mathcal{P}}$ is the natural substitution of $\{\theta' \mid \theta' \in [X \rightharpoonup V] \wedge \theta' \sqsubset \theta\}$. We have to show the following:*

(1) If there exists an $\mathcal{X}' \in S_{\mathcal{P}}$ with $\Delta(\theta \upharpoonright \mathcal{X}')$ defined, then there is an $\mathcal{X}'' \in S_B$ with $\Delta(\theta \upharpoonright \mathcal{X}'')$ defined.

(2) If there exists an $\mathcal{X}'' \in S_B$ with $\Delta(\theta \upharpoonright \mathcal{X}'')$ defined, then there is an $\mathcal{X}'' \in S_{\mathcal{P}}$ with $\Delta(\theta \upharpoonright \mathcal{X}')$ defined.

Proof. Note that $\mathcal{X}'$ and $\mathcal{X}''$ do not need to be identical. We can observe that $\text{BASE}_{\mathcal{P}}^X \subseteq \mathcal{P}_f(X)$, therefore, $S_B \subseteq S_{\mathcal{P}}$. S_B can be written as $S_{\mathcal{P}} \cap \text{BASE}_{\mathcal{P}}^X$.

(1): Let $\theta' = \theta \upharpoonright \mathcal{X}'$ with $\Delta(\theta')$ defined and $\mathcal{X}' \in S_{\mathcal{P}}$. If $\Delta(\theta')$ is defined, then it can either be defined in DEFINETO or DEFINENEW, since new definitions for Δ are created only there. If it is defined in DEFINENEW, then it can only be carried by a base event since the argument for DEFINENEW is the input event $e\langle\theta\rangle$, hence, the proposition holds. In the case it is defined in DEFINETO, then there must exist an instance $\theta'' \in [X \rightharpoonup V]$ already defined in Δ, since $\Delta(\theta') \leftarrow \Delta(\theta'')$ in Line 42. We know that monitor states are passed in a chain $\Delta(\theta_n) \leftarrow \cdots \leftarrow \Delta(\theta_0)$ with $\theta_0 \sqsubset \theta_n$. Because the first instance in the chain must already be defined in Δ, θ_0 must be an instance carried by a base event with $\Delta(\theta_0)$ defined in DEFINENEW, hence, $\text{DOM}(\theta_0) \in \text{BASE}_{\mathcal{P}}^X$. Assuming $\theta' = \theta_n$, it is sufficient to show that $\text{DOM}(\theta_0) \in S_{\mathcal{P}}$ considering $\text{DOM}(\theta_0) \in \text{BASE}_{\mathcal{P}}^X$ and $S_B = S_{\mathcal{P}} \cap \text{BASE}_{\mathcal{P}}^X$. We assume for contradiction that $\text{DOM}(\theta_0)$ is not in $S_{\mathcal{P}}$. If $\theta_0 \sqsubset \theta'$, then $\text{DOM}(\theta_0) \subset \text{DOM}(\theta')$, therefore, $\text{DOM}(\theta_0) \in \mathcal{P}_f(\text{DOM}(\theta'))$ and consequently $\text{DOM}(\theta_0) \in \mathcal{P}_f(\text{DOM}(\theta))$. The only possibility that $\text{DOM}(\theta_0)$ is not in $S_{\mathcal{P}}$ is by violating the constraint $\text{DOM}(\theta_0) \subset \text{DOM}(\theta)$ derived from the original definition of $S_{\mathcal{P}}$, i.e., $\{\mathcal{X} \mid \mathcal{X} \in \mathcal{P}_f(\text{DOM}(\theta)) \wedge \mathcal{X} \subset \text{DOM}(\theta)\}$. But the assumption of $\text{DOM}(\theta_0) \not\subset \text{DOM}(\theta)$ ultimately contradicts $\text{DOM}(\theta_0) \subset \text{DOM}(\theta')$ in conjunction with $\text{DOM}(\theta') \subset \text{DOM}(\theta)$, hence, $\text{DOM}(\theta_0)$ must be in $S_{\mathcal{P}}$.

(2): If $\Delta(\theta \upharpoonright \mathcal{X}'')$ defined, then $\Delta(\theta \upharpoonright \mathcal{X}')$ defined with some $\mathcal{X}' \in S_{\mathcal{P}}$ and some $\mathcal{X}'' \in S_B$. We assume for contradiction that there exists an $\mathcal{X}''$ in S_B, such that $\Delta(\theta \upharpoonright \mathcal{X}'')$ is defined, but no $\mathcal{X}' \in S_{\mathcal{P}}$ exists, such that $\Delta(\theta \upharpoonright \mathcal{X}')$ is defined. If $\mathcal{X}'' \in S_B$, then with $S_B \subseteq S_{\mathcal{P}}$ follows $\mathcal{X}'' \in S_{\mathcal{P}}$. Contradiction found, hence, the proposition holds. □

DEFINETO is sufficiently similar to $\text{DEFINETO}_{\mathbb{D}\langle X\rangle}$, so we only have to show the correctness of the optimized check for existing monitors as it deviates from the version in algorithm $\mathbb{D}\langle X\rangle$.

Proposition 4. *Let θ and $\mathcal{X}'$ be the input arguments of* DEFINETO*, $S_{\mathcal{P}}$ be the set $\{\mathcal{X} \mid \mathcal{X} \in \mathcal{P}_f(X) \wedge \mathcal{X} \subseteq \text{DOM}(\theta) \wedge \mathcal{X} \not\subseteq \mathcal{X}'\}$ and S_B be the set $\{\mathcal{X} \mid \mathcal{X} \in \text{BASE}_{\mathcal{P}}^X \wedge \mathcal{X} \subseteq \text{DOM}(\theta) \wedge \mathcal{X} \not\subseteq \mathcal{X}'\}$ used in the for-loop in algorithm $\mathbb{D}'\langle X\rangle$, line 37. If* DEFINETO *with the for-loop over $S_{\mathcal{P}}$ is observationally equivalent to $\text{DEFINETO}_{\mathbb{D}\langle X\rangle}$, then it is also observationally equivalent to* DEFINENEW *with the for-loop over S_B. We assume* DEFINETO *with the for-loop over $S_{\mathcal{P}}$ is observationally equivalent to $\text{DEFINETO}_{\mathbb{D}\langle X\rangle}$, because $S_{\mathcal{P}}$ is the natural substitution of $\{\theta'' \mid \theta'' \in [X \rightharpoonup V] \wedge \theta'' \sqsubseteq \theta \wedge \theta'' \not\sqsubseteq \theta'\}$ where $\theta' = \theta \upharpoonright \mathcal{X}'$.*

Let FAILTIMECHECK(θ_1, θ_2) : $([X \rightharpoonup V] \times [X \rightharpoonup V]) \rightarrow \{$**true**, **false**$\}$ *be a function defining the predicate in Line 38, i.e.,* FAILTIMECHECK$(\theta_1, \theta_2) = \mathcal{T}_\Theta(\theta_2) > \mathcal{T}_\Delta(\theta_1) \vee \mathcal{T}_\Delta(\theta_2) < \mathcal{T}_\Delta(\theta_1)$*, with $\mathcal{T}_\Theta$ and $\mathcal{T}_\Delta$ used in the algorithm. We have to show the following:*

(1) *If there exists an $\mathcal{X}_\mathcal{P} \in S_\mathcal{P}$ with* $\textsc{FailTimeCheck}(\theta, \theta \upharpoonright \mathcal{X}_\mathcal{P}) = \mathsf{true}$*, then there is an $\mathcal{X}_B \in S_B$ with* $\textsc{FailTimeCheck}(\theta, \theta \upharpoonright \mathcal{X}_B) = \mathsf{true}$*.*

(2) *If there exists a $\mathcal{X}_B \in S_B$ with* $\textsc{FailTimeCheck}(\theta, \theta \upharpoonright \mathcal{X}_B) = \mathsf{true}$*, then there is an $\mathcal{X}_\mathcal{P} \in S_\mathcal{P}$ with* $\textsc{FailTimeCheck}(\theta, \theta \upharpoonright \mathcal{X}_\mathcal{P}) = \mathsf{true}$*.*

Proof. $\mathcal{X}_\mathcal{P}$ and $\mathcal{X}_B$ do not need to be identical. Note that $\textsc{Base}^X_P \subseteq \mathcal{P}_f(X)$, therefore, $S_B \subseteq S_\mathcal{P}$, hence, $S_B = S_\mathcal{P} \cap \textsc{Base}^X_P$.

(1): Given θ and $\mathcal{X}'$ as arguments of $\textsc{DefineTo}$ with $\mathcal{X}_\mathcal{P} \in S_\mathcal{P}$ such that $\textsc{FailTimeCheck}(\theta, \theta \upharpoonright \mathcal{X}_\mathcal{P}) = \mathsf{true}$. It is sufficient to show $\mathcal{X}_\mathcal{P}$ is in $\textsc{Base}^X_P$, since in conjunction with $S_B = S_\mathcal{P} \cap \textsc{Base}^X_P$ follows $\mathcal{X}_\mathcal{P} \in S_B$.

The time check is a disjunction of two predicates. We will show the validity of the proposition for the two predicates in the disjunction separately.

Assume that the left side of the conjunction, $\mathcal{T}_\Theta(\theta \upharpoonright \mathcal{X}_\mathcal{P}) > \mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}')$, evaluates to true. The only definition of $\mathcal{T}_\Theta$ happens in Line 7 with $\mathcal{T}_\Theta(\theta_i)$, where θ_i is the input instance. Since θ_i is the input instance, $\textsc{Dom}(\theta_i)$ must be in $\textsc{Base}^X_P$. With $\theta_i = \theta \upharpoonright \mathcal{X}_\mathcal{P}$, implying $\textsc{Dom}(\theta \upharpoonright \mathcal{X}_\mathcal{P}) = \mathcal{X}_\mathcal{P}$ is in $\textsc{Base}^X_P$, the proposition holds for the left side of the conjunction.

Assume that $\mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}_\mathcal{P}) < \mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}')$, being the right side of the conjunction in $\textsc{FailTimeCheck}$, evaluates to true. Because $\Delta(\theta \upharpoonright \mathcal{X}')$ is defined, $\mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}') < \infty_+$, therefore, $\mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}_\mathcal{P}) < \infty_+$, otherwise, $\mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}_\mathcal{P}) < \mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}') = \mathsf{false}$, which would contradict the assumption. Hence, $\mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}_\mathcal{P})$ is either defined in $\textsc{DefineNew}$ or in $\textsc{DefineTo}$. If it is defined in $\textsc{DefineNew}$, then it is defined by an input instance θ_i which is associated to a base event and $\textsc{Dom}(\theta_i) \in \textsc{Base}^X_P$. With $\theta_i = \theta \upharpoonright \mathcal{X}_\mathcal{P}$ the proposition holds. In the other case it is defined in $\textsc{DefineTo}$. If it is defined in $\textsc{DefineTo}$, then there must exist an instance $\theta' \in [X \rightharpoonup V]$ already defined in $\mathcal{T}_\Delta$, since $\mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}_\mathcal{P}) \leftarrow \Delta(\theta')$ in Line 42. We know that the timestamps are passed each time monitor states are passed, forming a chain $\mathcal{T}_\Delta(\theta_n) \leftarrow \cdots \leftarrow \mathcal{T}_\Delta(\theta_0)$ with $\theta_0 \sqsubset \theta_n$. Because the first instance in the chain must have been carried by a base event, we conclude $\textsc{Dom}(\theta_0) \in \textsc{Base}^X_P$. Assuming $\theta \upharpoonright \mathcal{X}_\mathcal{P} = \theta_n$, it is sufficient to show that $\textsc{Dom}(\theta_0) \in S_\mathcal{P}$, considering $\textsc{Dom}(\theta_0) \in \textsc{Base}^X_P$ and $S_B = S_\mathcal{P} \cap \textsc{Base}^X_P$. We assume for contradiction that $\textsc{Dom}(\theta_0)$ is not in $S_\mathcal{P}$. Since $\textsc{Dom}(\theta_0) \subseteq \mathcal{P}_f(X)$, the only possibility that $\textsc{Dom}(\theta_0)$ is not in $S_\mathcal{P}$ is by violating the constraint $\textsc{Dom}(\theta_0) \subseteq \textsc{Dom}(\theta) \wedge \textsc{Dom}(\theta_0) \not\subseteq \mathcal{X}'$ derived from the original definition of $S_\mathcal{P}$ which is $\{\mathcal{X} \mid \mathcal{X} \in \mathcal{P}_f(X) \wedge \mathcal{X} \subseteq \textsc{Dom}(\theta) \wedge \mathcal{X} \not\subseteq \mathcal{X}'\}$. If $\theta_0 \sqsubset \theta \upharpoonright \mathcal{X}_\mathcal{P}$, then $\textsc{Dom}(\theta_0) \subset \textsc{Dom}(\theta \upharpoonright \mathcal{X}_\mathcal{P})$ with $\textsc{Dom}(\theta \upharpoonright \mathcal{X}_\mathcal{P}) \subseteq \textsc{Dom}(\theta)$, hence, it is impossible to violate $\textsc{Dom}(\theta_0) \subseteq \textsc{Dom}(\theta)$. We, therefore, assume for contradiction that the negated other part of the conjunction, namely $\textsc{Dom}(\theta_0) \subseteq \mathcal{X}'$, holds. We know that $\theta_0 \sqsubset \theta \upharpoonright \mathcal{X}_\mathcal{P}$, hence, $\theta_0 \sqsubset \theta$. Combined with $\textsc{Dom}(\theta_0) \subseteq \mathcal{X}'$, we can derive $\theta_0 \sqsubset \theta \upharpoonright \mathcal{X}'$. Both $\Delta(\theta_0)$ and $\Delta(\theta \upharpoonright \mathcal{X}')$ are defined, which implies with $\theta_0 \sqsubset \theta \upharpoonright \mathcal{X}'$ that there must be a chain of passed timestamps $\mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}') \leftarrow \cdots \leftarrow \mathcal{T}_\Delta(\theta_0)$. Consequently, $\mathcal{T}_\Delta(\theta_0) = \mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}')$. Because $\mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}_\mathcal{P}) \leftarrow \cdots \leftarrow \mathcal{T}_\Delta(\theta_0)$ and the initial assumption $\mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}_\mathcal{P}) < \mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}')$, follows $\mathcal{T}_\Delta(\theta_0) < \mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}')$, which contradicts $\mathcal{T}_\Delta(\theta_0) = \mathcal{T}_\Delta(\theta \upharpoonright \mathcal{X}')$. Hence, the proposition holds.

(2): If $\Delta(\theta \upharpoonright \mathcal{X}'')$ defined, then $\Delta(\theta \upharpoonright \mathcal{X}')$ defined with some $\mathcal{X}' \in S_\mathcal{P}$ and some $\mathcal{X}'' \in S_B$. We assume for contradiction that there exists an $\mathcal{X}_B \in S_B$, such that $\textsc{FailTimeCheck}(\theta, \theta \upharpoonright \mathcal{X}_B) = \mathsf{true}$, but no $\mathcal{X}_\mathcal{P} \in S_\mathcal{P}$ exists, such that $\textsc{FailTimeCheck}(\theta, \theta \upharpoonright \mathcal{X}_\mathcal{P}) = \mathsf{true}$. If $\mathcal{X}_B \in S_B$, then with $S_B \subseteq S_\mathcal{P}$ follows $\mathcal{X}_B \in S_\mathcal{P}$. Contradiction found, so the proposition holds. □

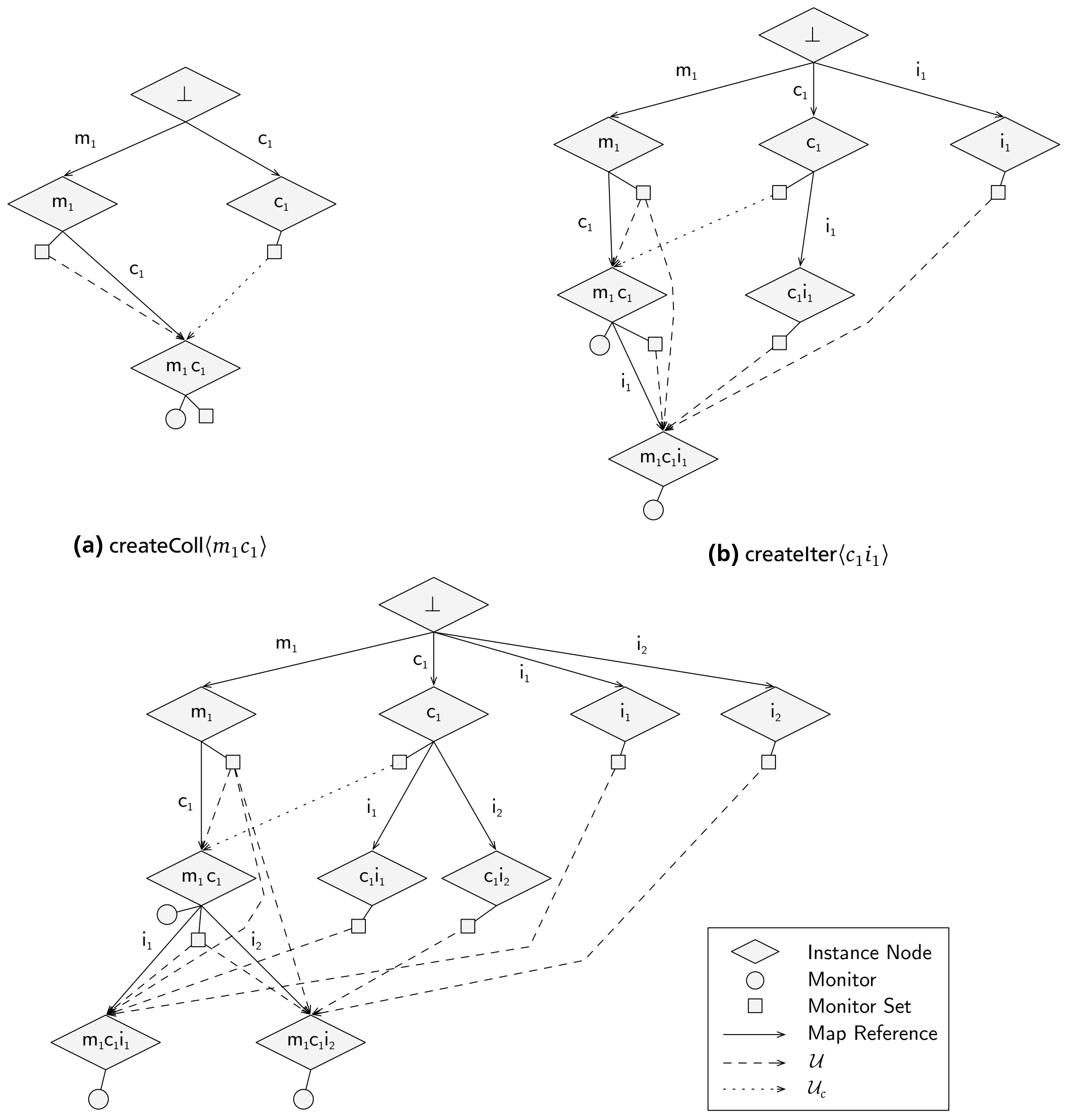
3.1.3 Optimizations

Many data structures used in algorithm $\mathbb{D}'\langle X\rangle$ are functions depending on instances. The value $\Delta(\theta)$ represents the current monitor state for an instance $\theta \in [X \rightharpoonup V]$. $\mathcal{T}_\Theta(\theta)$ denotes the last time the instance was seen and $\mathcal{T}_\Delta(\theta)$ represents the time a monitor was created. The functions $\mathcal{U}$ and $\mathcal{U}_c$ map to sets depending on instances more informative then θ. A fast and memory efficient data structure is, therefore, necessary to store and retrieve instances and associated data on demand.

A data structure for instance indexing has to exploit the characteristic structure of parametric instances as they are represented in the system. As defined in Definition 4, an instance θ can be considered a sequence of mappings of parameters to parameter values $\langle x_i \mapsto v_j, \ldots, x_k \mapsto v_l\rangle$. Let $X = \{a, b, c\}$ and $V = \{m_1, c_1, c_2, i_1, i_2, i_3\}$. The instances $\langle m_1 c_1 i_1\rangle$, $\langle m_1 c_1 i_2\rangle$ and $\langle m_1 c_1 i_3\rangle$ can all be described as sequences having the same prefix $m_1 c_1$ differing only in their suffix i_1, i_2 or i_3. It is efficient to store this prefix only once in the data structure. A data structure with the desired property is a *multi-dimensional associative array*, or *multi-dimensional map*. A map associates keys to values. In a multi-dimensional map the values themselves are maps, mapping to other maps, allowing deep nested indexing structures. Using a three-dimensional map, each parameter value in an instance can be used as keys to access the next map. Starting from a root map, the m_1-key is associated to a m_1-map, which associates the c_1-key to a $m_1 c_1$-map. From there, the three key i_1, i_2 and i_3 map to values representing the instances $\langle m_1 c_1 i_1\rangle$, $\langle m_1 c_1 i_2\rangle$ and $\langle m_1 c_1 i_3\rangle$.

The multi-dimensional map with parameter values as keys is the foundation for the simplified data structure used in this section. This data structure is called *indexing tree*. The nodes in this tree are *instance nodes* and store the timestamp $\mathcal{T}_\Theta(\theta)$, a monitor storing the monitor state, the timestamp $\mathcal{T}_\Delta(\theta)$ and a collection of *monitor sets*. A monitor set holds references to instance nodes which are relevant for monitor state updates or for compatible instance retrieval. In other words, the monitor sets of an instance node for an instance θ realize the functions $\mathcal{U}(\theta)$ and $\mathcal{U}_c(\theta, \mathcal{X})$ for relevant $\mathcal{X} \subseteq X$. We say that a node fully *instantiates* an instance θ, if the node is reached by traversing the index tree using all bindings in the instance. Therefore, a node that fully instantiates an instance also represents it.

Figure 3.6 shows the data structure with algorithm $\mathbb{D}'\langle X \rangle$ consuming a trace specified by the UnsafeIteratorProperty P_{umi^*}. The trace createColl$\langle m_1 c_1 \rangle$ createIter$\langle c_1 i_1 \rangle$ createIter$\langle c_1 i_2 \rangle$ represents the creation of a collection based on a map followed by the creation of two iterators based on the collection. As depicted in the Figure 3.6, the processing of each event in this seemingly simple trace requires the creation of multiple objects and references. The processing of future events of type updateMap$\langle m_1 \rangle$ and useIter$\langle i_1 \rangle$ or useIter$\langle i_2 \rangle$ can depend on this structure without any further object creation. Note that the number of monitor sets depends on the property. When monitoring with the UnsafeMapIterator property each node can have at most one monitor set. The data structure is simplified as it does not take into account information needed to execute code on trace matching or maintenance needed to minimize memory consumption. The description of the detailed data structure can be found in Section 3.2.



(a) createColl$\langle m_1 c_1 \rangle$

(b) createIter$\langle c_1 i_1 \rangle$

(c) createIter$\langle c_1 i_2 \rangle$

Figure 3.6.: Simplified indexing tree for the trace createColl$\langle m_1 c_1 \rangle$ createIter$\langle c_1 i_1 \rangle$ createIter$\langle c_1 i_2 \rangle$

3.1.3.1 Merging $\mathcal{U}$ and $\mathcal{U}_c$

As introduced in algorithm $\mathbb{D}'\langle X\rangle$, $\mathcal{U}$ and $\mathcal{U}_c$ are used for different purposes. The former for performing updates of monitor states, the latter, to keep references to compatible instances considered as potential combination partners for incoming instances. Figure 3.7 shows an example with instance $\theta = \langle a_1\rangle$ and $\mathcal{U}$ and $\mathcal{U}_c$ mapping to the more informative instances $\langle a_1 b_1\rangle, \langle a_1 b_1 c_1\rangle, \langle a_1 b_1 c_2\rangle, \langle a_1 b_1 c_1 d_1\rangle$ and $\langle a_1 b_1 c_1 d_2\rangle$. All more informative instances have to be updated on events carrying instance a_1, but compatible instances are only retrieved for the domain $\{a, b\}$.

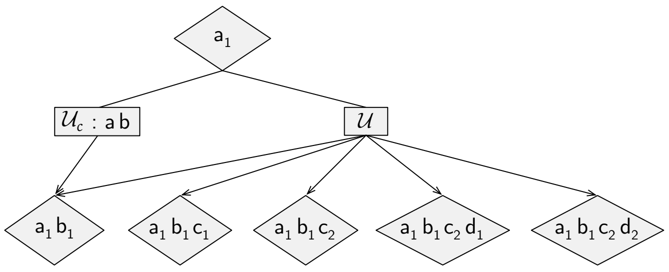


Figure 3.7.: Monitor sets using separate functions $\mathcal{U}$ and $\mathcal{U}_c$ as defined in algorithm $\mathbb{D}'\langle X\rangle$

As can be shown, maintaining two functions is not necessary and the functions can be merged into one. It is known that $\theta' \in \mathcal{U}(\theta)$ with $\theta \sqsubset \theta'$ and $\theta'' \in \mathcal{U}_c(\theta, \mathcal{X})$ with $\theta \sqsubset \theta''$ and $\textsc{Dom}(\theta'') = \mathcal{X}$ holds for all $\theta \in [X \rightharpoonup V]$ and all $\mathcal{X} \in \mathcal{P}_f(X)$. We can conclude from $\textsc{Dom}(\theta'') = \mathcal{X}$ that all defined sets $\mathcal{U}_c(\theta, \mathcal{X})$ for a fixed $\theta \in [X \rightharpoonup V]$ are disjunct over all $\mathcal{X} \in \mathcal{P}_f(X)$.

Proposition 5. *Given $\mathcal{U}_c$ the function used in algorithm $\mathbb{D}'\langle X\rangle$ then $\cap\{\mathcal{U}_c(\theta,\mathcal{X}) \mid \theta \in [X \rightharpoonup V] \wedge \mathcal{X} \in \mathcal{P}_f(X) \wedge \mathcal{U}_c(\theta,\mathcal{X})$ defined$\} = \emptyset$.*

Proof. Let θ be an instance in $[X \rightharpoonup V]$ and $\mathcal{X}, \mathcal{X}'$ sets in $\mathcal{P}_f(X)$ with $\mathcal{X} \neq \mathcal{X}'$. Assume for contradiction that there exists an $\theta'' \in [X \rightharpoonup V]$ so that $\theta'' \in \mathcal{U}_c(\theta, \mathcal{X})$ and $\theta'' \in \mathcal{U}_c(\theta, \mathcal{X}')$, implying $\mathcal{U}_c(\theta,\mathcal{X}) \cap \mathcal{U}_c(\theta,\mathcal{X}') \neq \emptyset$. Then $\textsc{Dom}(\theta'') = \mathcal{X}$ and $\textsc{Dom}(\theta'') = \mathcal{X}'$. Hence $\mathcal{X} = \mathcal{X}'$, contradiction found, therefore, the sets must be disjunct. □

If the sets stored in $\mathcal{U}_c(\theta, \mathcal{X})$ for all $\mathcal{X} \in \mathcal{P}_f(X)$ are disjunct, it is viable to use $\mathcal{U}_c$ in MAIN to perform update operations. Instead of updating the monitor state of $\Delta(\theta')$ for all $\theta' \in \{\theta' \mid \theta' \in \mathcal{U}(\theta) \wedge \Delta(\theta')$ defined$\}$, the update can iterate over all sets in $\mathcal{P}_f([X \rightharpoonup V])$ which are associated with θ in $\mathcal{U}_c$.

Therefore, the update for an incoming $e\langle\theta\rangle$ can be performed with the following loop.

```
for all $\mathcal{X}' \in \{\mathcal{X}' \mid \mathcal{X}' \in \mathcal{P}_f(X) \wedge (\theta, \mathcal{X}') \in \textsc{Dom}(\mathcal{U}_c)\}$ do
    for all $\theta' \in \mathcal{U}_c(\theta, \mathcal{X}')$ s.t. $\Delta(\theta')$ defined do
        $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$
    end for
end for
```

Note that if proposition 5 would be false, the above solution would not be correct. If an instance θ' would be contained in multiple sets, its monitor would also process the base event e multiple times, leading to incorrect traces. In that case we would need to perform the update over the union $\cup\{\mathcal{U}_c(\theta,\mathcal{X}) \mid \theta \in [X \rightharpoonup V] \wedge \mathcal{X} \in \mathcal{P}_f(X) \wedge \mathcal{U}_c(\theta,\mathcal{X})$ defined$\}$, which would require its creation on each incoming event. Note that retrieving the domain of $\mathcal{U}_c$ in the indexing tree is simple, because θ represents an instance node. We, therefore, just need to iterate over all monitor sets stored in this node.

Using only $\mathcal{U}_c$ to realize monitor updates, the instances formerly added to $\mathcal{U}$ have now to be placed into $\mathcal{U}_c$ while passing the domain of the instance. The assignment $\mathcal{U}(\theta') \leftarrow \mathcal{U}(\theta') \cup \{\theta\}$ becomes $\mathcal{U}_c(\theta', \textsc{Dom}(\theta)) \leftarrow \mathcal{U}_c(\theta', \textsc{Dom}(\theta)) \cup \{\theta\}$. However, this solution has a downside. The less informative instance θ needs to hold a monitor set for each different instance domain of all more informative instances. Applied to the example above, the instance node for a_1 would need a monitor set for each of the domains $\{a, b\}$, $\{a, b, c\}$ and $\{a, b, c, d\}$. This is correct even if the instances are only updated and not retrieved as compatible combination partners. Because this is not efficient, an optimized solution is proposed. Since the compatible instance searched from θ has the domain $\{a, b\}$ and the other more informative instances are only accessed during the update phase, there is no reason for the partitioning of the instances into three sets. Consequently, $\{a, b, c\}$ and $\{a, b, c, d\}$ can all be merged into one set. Note that, because the sets are disjunct, this solution is compatible with the need to reference specific instances. A merged monitor set can be maintained in parallel to sets holding instances with a specific domain.

A mere technical problem is posed by the original semantics of parameters of $\mathcal{U}_c$. The parameter $\mathcal{X}$ represents a parameter set, denoting the domain of all instances to be contained in the set. This semantics have to be altered for merged monitor sets. To express that an instance has to be placed into the merged monitor set, a marker set is used, signifying that this set does not care about the domains of its instances. We have chosen the empty set as a marker set, because there can not be any $\theta' \in \mathcal{U}_c(\theta, \mathcal{X})$ with $\textsc{Dom}(\theta') = \emptyset$ without contradicting $\theta' \sqsubset \theta$.

Figure 3.8 shows the merged monitor set used in parallel to a set referencing compatible instances. Note that each instance is only referenced by one set, as compared to Figure 3.7.

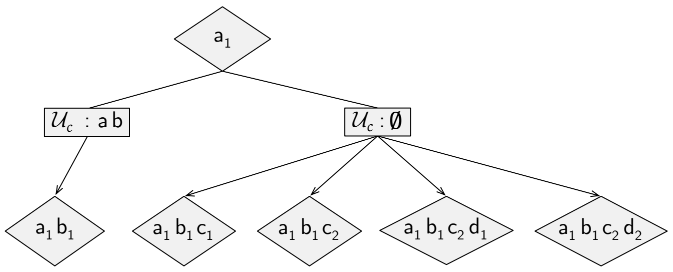


Figure 3.8.: Monitor sets with function $\mathcal{U}$ merged into $\mathcal{U}_c$

3.1.3.2 Preventing Non-State-Changing Updates

It is necessary to define an ordering over the set of parameters, otherwise it is not definite in which order the bindings have to be used to traverse the maps. If the order would be arbitrary, there could be multiple maps for the same instance. To formalize the order used in the multi-dimensional indexing, we introduce the *parameter tree*:

Definition 25. ***(Parameter tree)*** *Given the set of parameters X, let $\prec_X$ be a strict total order over X. An parameter tree is a graph (V_X, E_X) with a set of nodes $V_X \subseteq \mathcal{P}_f(X)$ and a set of edges $E_X \subseteq \mathcal{P}_f(X) \times \mathcal{P}_f(X)$, connecting each node in $\mathcal{P}_f(X)$. For each edge $(\mathcal{X}, \mathcal{X}') \in E_X$ the property $\mathcal{X}' = \mathcal{X} \cup \{x'\}$ with $x' \in X \setminus \mathcal{X}$ and $x \prec_X x'$ for all $x \in \mathcal{X}$ holds.*

Given a set of parameters X this constructs an oriented tree (or polytree) with the empty set as root. Note that for a set of parameters X with a given order $\prec_X$ there exists exactly one parameter tree which instantiates all nodes in $\mathcal{P}_f(X)$, the *full parameter tree for* X. Figure 3.9 shows a full parameter tree using the set of parameters $X = \{a, b, c\}$. Each node denotes a parameter set $\mathcal{X} \subseteq X$, which represents an instance θ with $\textsc{Dom}(\theta) = \mathcal{X}$.

$$\begin{tikzcd}
\emptyset \arrow[d] \arrow[dr] \arrow[drr] & & \\
\{a\} \arrow[d] \arrow[dr] & \{b\} \arrow[d] & \{c\} \\
\{a,b\} \arrow[d] & \{a,c\} \quad \{b,c\} & \\
\{a,b,c\} & &
\end{tikzcd}$$

Figure 3.9.: Full parameter tree for $X = \{a, b, c\}$ with $a \prec_X b \prec_X c$

We can extend the parameter tree with additional information regarding instance monitors, base events and necessary mappings in the $\mathcal{U}_c$-function. Figure 3.10 shows the instance tree for the UnsafeMapIterator property P_{umi^*} with an overlaying graph visualizing necessary mappings in $\mathcal{U}_c$ with dashed edges. These edges are called $\mathcal{U}_c$*-edges*. The e in the upper right corner of a set $\mathcal{X}$ denotes that there exists an event $e \in \mathcal{E}$ with $\mathcal{D}_{\mathcal{E}}(e) = \mathcal{X}$. The Δ-symbol in the lower right corner expresses that a monitor is created for each instance of type $\mathcal{X}$. As described in Section 3.1.3.1, $\mathcal{U}_c$-mappings are needed for two purposes: 1. When an event is encountered, all monitors in its monitor set will be notified to update their monitor states. In the figure, this are all parameter nodes which are marked with an e, notifying the two nodes marked with Δ. 2. In the join phase, compatible instances are found by retrieving more informative instances of a strictly less informative instance. E.g., on event $\mathsf{createIter}\langle c_1 i_1 \rangle$, due to the enable set $\textsc{PartEnable}^{X}_{\mathcal{G}}(\mathsf{createIter}) = \{\{m, c\}\}$ the more informative instances of $\langle c_1 \rangle$ with domain $\{m,c\}$ are examined, to find the matching $\langle m_j c_1 \rangle$. The edge $\{c\} \mapsto \{m,c\}$ represents all $\mathcal{U}_c$-mappings necessary to access compatible instances for combinations with instances with domain $\{c,i\}$.

As can be observed from Figure 3.10, the node $\{m,i\}$ is missing in the parameter tree, although $\{m,i\} \subset \mathcal{P}_f(X)$. This is due to its irrelevance for the monitoring result. Since $\{m,i\}$ is a leaf in the tree, it is also not necessary for reaching other nodes while indexing, as is, e.g., $\{c\}$. It can be concluded that leaf nodes which do not carry monitors and are not associated to base events can be pruned from the parameter tree. Note that this pruning can be performed incrementally to remove branches node by node. A fully pruned tree is called *effective parameter tree*. Which instances carry monitors can be derived from the enable sets and creation events.

Proposition 6. *Let* $\Delta : [X \rightharpoonup V] \rightharpoonup S$ *be the function in algorithm* $\mathbb{D}'\langle X \rangle$*, then* $\Delta(\theta)$ *can be only defined for instances* $\theta \in [X \rightharpoonup V]$ *with* $\textsc{Dom}(\theta) \in \{\mathcal{X}_e \cup \mathcal{D}_{\mathcal{E}}(e) \mid \mathcal{X}_e \in \textsc{PartEnable}_{\mathcal{G}}^{X}(e) \wedge e \in \mathcal{E}\} \cup \{\mathcal{D}_{\mathcal{E}}(e) \mid e \in \textsc{Create}_{\mathcal{P}}^{\mathcal{E}}\}$.

Proof. Let $e\langle\theta\rangle \in \mathcal{E}\langle X \rangle$ be the current input event, and let $\theta' \in [X \rightharpoonup V]$ be the instance defining $\Delta(\theta')$ in algorithm $\mathbb{D}'\langle X \rangle$. $\Delta(\theta')$ can only be defined in $\textsc{DefineNew}$ and $\textsc{DefineTo}$. If it is defined in $\textsc{DefineNew}$ then it is defined by the instance carried by the input event, hence, $\theta' = \theta$, implying $\textsc{Dom}(\theta') = \textsc{Dom}(\theta) = \mathcal{D}_{\mathcal{E}}(e)$. Since $\textsc{DefineNew}$ is only called when the base event e is a creation event, $\mathcal{D}_{\mathcal{E}}(e)$ must be in $\{\mathcal{D}_{\mathcal{E}}(e) \mid e \in \textsc{Create}_{\mathcal{P}}^{\mathcal{E}}\}$. If $\Delta(\theta')$ is defined in $\textsc{DefineTo}$ then it is called from $\textsc{CreateNewMonitorStates}$. If it is called from Line 16 then $\theta' = \theta$, implying $\textsc{Dom}(\theta') = \textsc{Dom}(\theta)$. We know from the constraints in Line 14 that $\mathcal{X}_e \subset \textsc{Dom}(\theta)$ with $\mathcal{X}_e \in \textsc{PartEnable}_{\mathcal{G}}^{X}(e)$. Therefore, $\textsc{Dom}(\theta') = \textsc{Dom}(\theta) = \mathcal{X}_e \cup \textsc{Dom}(\theta) = \mathcal{X}_e \cup \mathcal{D}_{\mathcal{E}}(e)$, letting the proposition hold. If $\textsc{CreateNewMonitorStates}$ is called from Line 22 then $\theta' = \theta \sqcup \theta_j$ with $\textsc{Dom}(\theta_j) \in \textsc{PartEnable}_{\mathcal{G}}^{X}(e)$. Follows, $\textsc{Dom}(\theta') = \mathcal{D}_{\mathcal{E}}(e) \cup \mathcal{X}_e$ with $\mathcal{X}_e \in \textsc{PartEnable}_{\mathcal{G}}^{X}(e)$. □

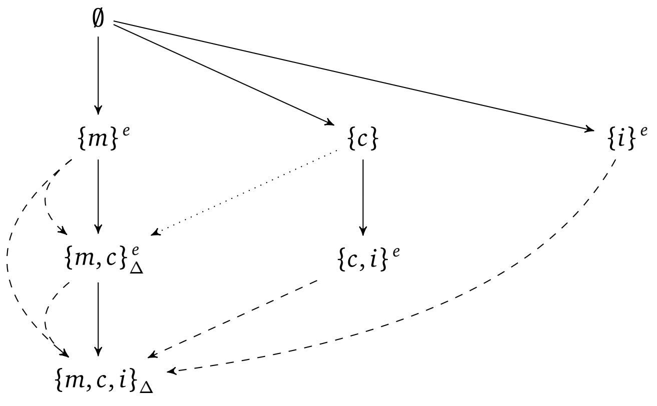


Figure 3.10.: Effective parameter tree for UnsafeMapIterator with $\mathcal{U}_c$-mappings (dashed lines) and $m \prec_X c \prec_X i$

An annotated effective parameter tree fully specifies which $\mathcal{U}_c$-mappings need to be created for concrete instances. It can be shown, that some $\mathcal{U}_c$-edges between instances in the tree can be removed by analysis of the property, without affecting the correctness of the monitoring result. An example of such an optimization can be found in UnsafeMapIterator. Recall the pattern constituting the UnsafeMapIterator specified in the FSM in Figure 3.1. As can be observed, the state s_1 lies on all paths to the accepting state starting with a creation event. No such path to the accepting state exists, which does not introduce the same set of parameters on state s_1, namely $\{m,c\}$. Hence, for each instance $\theta \in [X \rightharpoonup V]$ with $\textsc{Dom}(\theta) = \{m,c\}$ the equality $\Delta(\theta) = s_1$ holds. The FSM specifies that updateMap events on state s_1 are self-looping. Therefore, all base events updateMap$\langle\theta'\rangle$ with $\theta' \in [X \rightharpoonup V]$ updating $\Delta(\theta)$ with $\textsc{Dom}(\theta) = \{m,c\}$ do not change the monitor state of $\Delta(\theta)$. Hence, the removal of the $\mathcal{U}_c$-edge $\{m\} \mapsto \{m,c\}$ does not affect correctness of the monitoring result. This simplification has the effect that no $\mathcal{U}_c$-mappings from $\langle m_j \rangle \mapsto \langle m_j c_k \rangle$ have to be created for events with base event createColl$\langle m_j c_i \rangle$. Consequently, no instances $\langle m_j c_k \rangle$ need to updates their monitor states on events updateMap$\langle m_k \rangle$.

This optimization can be generalized as follows. Firstly, all associations of parameter sets to states have to be found. Then all $\mathcal{U}_c$-mappings from base events to each of those parameter sets which do not change any of the states can be removed.

3.2 Data Structure

In this section we introduce a data structure for implementation of algorithm $\mathbb{D}'\langle X \rangle$ targeting platforms with *garbage collection* and *weak references*. This data structure is used as the foundation for our reference implementation described in Chapter 4.

Garbage collection [26] is known as automatic memory management where unused objects are reclaimed by a background process called *garbage collector*. A tracing garbage collector, as implemented in most platforms including Java, discerns used an unused objects by their direct and indirect reachability via references from a root set of objects. If an object is not reachable from objects in the root set then it impossible to be used in the future, therefore, it can be removed from memory. The references pointing to objects preventing garbage collection are also known as *strong references*, hence, objects reachable via strong references are called *strongly reachable*. Weak references, on the other hand, are references pointing to objects which can be used like ordinary references but keep the object eligible for garbage collection. Therefore, by pointing a weak reference to an object its lifetime is not prolonged. Weakly reachable object are objects which are not strongly reachable but still can be reached from a weak reference. Since such objects are eligible for garbage collection they are usually reachable only for some time until their memory is reclaimed by the garbage collector.

3.2.1 Data Structure Overview

The data structure presented in this section is an extension of the simplified data structure described in Section 3.1.3. The known entities are instance node, monitor, monitor set and map reference. The map reference is described as the usage of the parameter value as key in the multi-dimensional map to access other instance nodes. We now introduce *bindings* and *node references* as additional objects.

A binding is an object which encapsulates a parameter value by pointing a weak reference to it. Bindings are needed to represent parameter values in the indexing tree without prolonging their lifetime. They are created, as events are generated in the monitored application carrying parameter values. A binding is created only once for a parameter value and is reused on each subsequent encounter of the same parameter value. Instead of parameter values, bindings can be used as keys in node maps. If the parameter value ceases to be strongly reachable, this can be detected and the binding is *expired*. Expired bindings can trigger maintenance measures in the indexing data structure releasing resources which became obsolete after binding expiration.

A node reference provides a weak reference to the instance node and a strong reference to a monitor. Therefore, the appropriate name would be *node and monitor reference* but we use the former for brevity. The node reference allows to retrieve nodes which use expired bindings as keys in node maps to prune the associated branches in the indexing tree. Because the reference to the node is weak, this guarantees that the lifetime of nodes in already pruned branches is not prolonged.

Table 3.3 summarizes the objects and roles in the data structure.

Object	Role
Instance node	Instantiates an instance, has a node map, has a number of monitor sets, stores the $\mathcal{T}_{\Theta}$ timestamp
Monitor	Stores the monitor state for an instance, stores the $\mathcal{T}_{\Delta}$ timestamp
Node reference	Weakly references a node, strongly references a monitor
Monitor set	Strongly references a number of node references
Binding	Weakly references a parameter value
Parameter value	An object in the monitored application

Table 3.3.: Objects and their roles in the data structure

The reference structure is depicted in Figure 3.11. Multiplicities are annotated, as they illustrate interesting characteristics of the data structure.

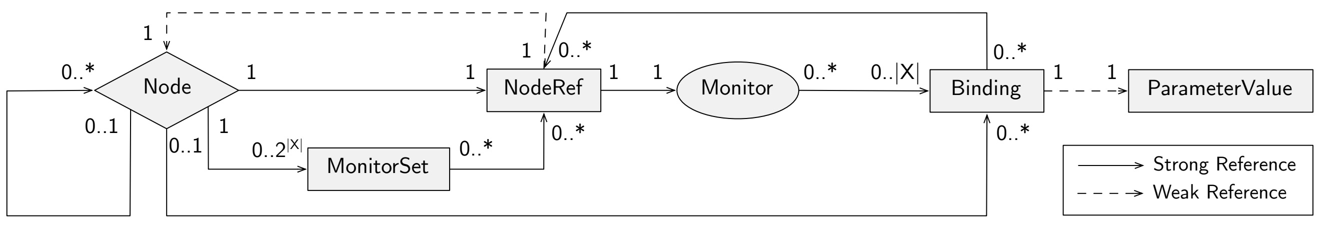


Figure 3.11.: Reference structure with multiplicities

The root node in the indexing tree has no ancestors. An instance node can map to an arbitrary number of nodes, or to no nodes at all if it is leaf in the indexing tree. An instance node, its node reference and its monitor are all bijectively associated. The monitor has to be reached from the instance node via the node reference. The number of monitor sets is limited by the number of different parameter sets which may be necessary to access compatible instances efficiently.

As the number of different parameter sets is bounded by the size of $\mathcal{P}_f(X)$, a node cannot have more then $2^{|X|}$ monitor sets. A monitor set can reference arbitrary numbers of node references each providing access to a monitor. There may be node references which are not referenced by any monitor set, implying the associated monitors do not receive updates from less informative nodes. A monitor stores at most $|X|$ bindings because each parameter can be only bound once in a single trace slice. A binding can be referenced by multiple monitors since each monitor has to store the bindings bound to the parameters in the monitor instance to be able to react with recovery code on matches. A node reference can be referenced by multiple bindings because the same binding may be used as a key in multiple bindings. Note that a node itself is a map entry, so each node which is not the root node does reference exactly one binding. As this can be considered an implementation detail, the interested reader is referred to Section 4.3.5 for more information.

Figure 3.12 shows an concrete example of an indexing tree branch with complete reference structure. We assume $X = \{a, b, c\}$, $V = \{a_1, b_1, c_1\}$ and $\mathcal{U}_c = \{(\langle a_1\rangle, \emptyset) \mapsto \{\langle a_1 b_1\rangle, \langle a_1 b_1 c_1\rangle\}, (\langle a_1 b_1\rangle, \emptyset) \mapsto \{\langle a_1 b_1 c_1\rangle\}\}$.

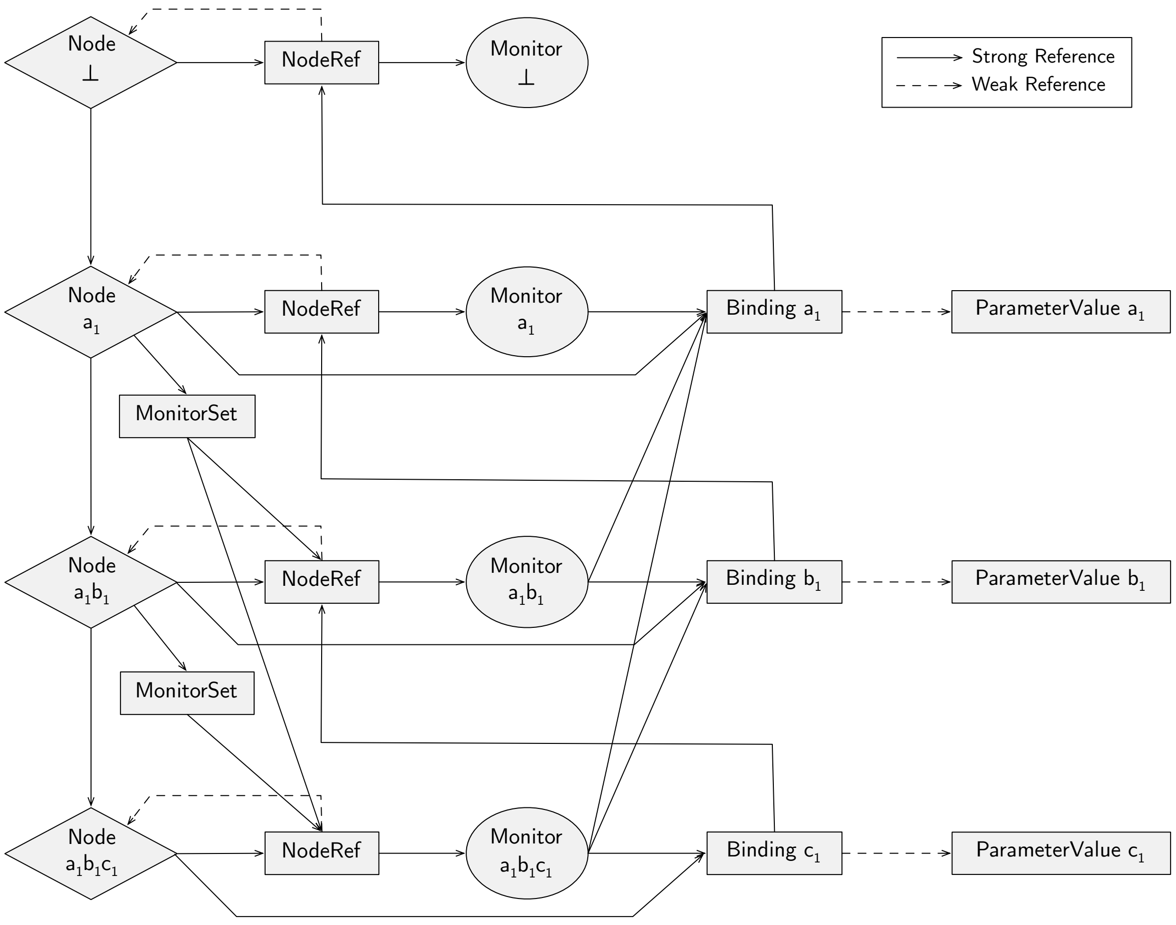
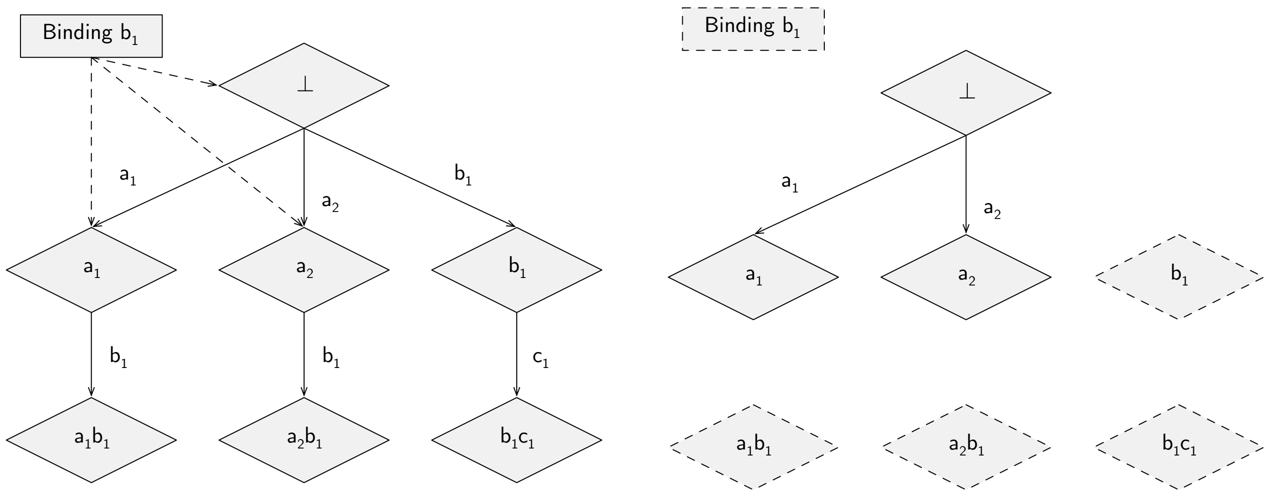


Figure 3.12.: Branch of an indexing tree with complete reference structure

It may look like an inefficiency that the bindings can be reached indirectly from the nodes via node reference and monitor as well as via direct reference. Note that there are cases where nodes do not have monitors, e.g., when they do not instantiate any instances themselves but are only "intermediate stations" for instantiation of more informative instances. In this cases the direct reference is the only reference. In all other cases reaching the binding directly should also be advised to be used for improved runtime performance, since the reference has to be followed on each traversal in the indexing tree.

3.2.2 Memory Management

When the parameter value weakly referenced by the binding is not strongly reachable anymore, it means that the parameter value will not be part of an input instance in the future. Therefore, the binding will not be used as a key in any of the node maps which use it. All branches in the indexing tree belonging to this key are, therefore, obsolete and have to be removed to prevent leaking memory. Therefore, a binding holds weak references to all nodes which use it as keys in node maps. Each time a binding is used as a key to create a new node in the indexing tree, it stores a weak reference to the node, namely, the node reference. When a binding expires, it accesses its nodes and removes all entries from the node maps which use the binding as key, as depicted in Figure 3.13.



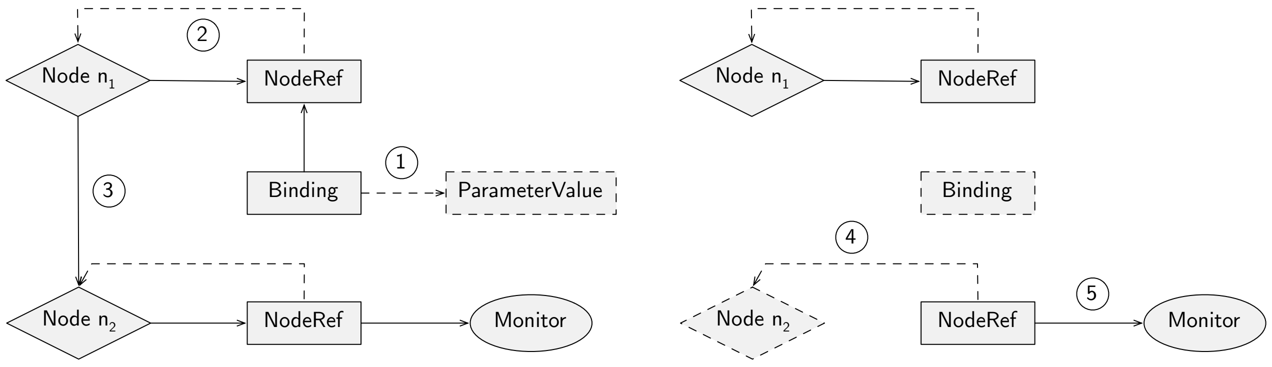
(a) References to node maps using b_1 as key

(b) Pruning of branches after expiration of b_1

Figure 3.13.: Pruning of branches in the index tree after expiration of a binding (simplified)

The expiration of the binding expresses that certain events are not expected in the future carrying instances with mappings to the now unreachable parameter value. This has also consequences for monitors as they may depend on these events for reaching an accepting state. An monitor which is unable to reach an accepting state because of unreachable parameter values is called a *dead monitor*. Dead monitors use resources without having the chance to report a match. Therefore, it is advised to try to check the aliveness of the monitor and remove it, if appropriate. As depicted in Figure 3.14, the collection of obsolete objects consists of two phases. In the first phase a binding expires forcing pruning in the indexing tree. Note that multiple branches may be pruned as the binding may be used in multiple node maps. In the second phase node references detect collected nodes and trigger an expiration check for the associated monitor. The steps in Figure 3.14 are as follows.

1. The binding detects that the parameter value is not strongly reachable and triggers indexing tree cleanup.
2. The node reference for node n_1 initiates pruning of the branch accessible with the binding as key.
3. The branch is pruned, and node n_2 is no longer strongly reachable.
4. The node reference of node n_2 detects the node is no longer strongly reachable.
5. The node reference checks aliveness of the monitor and removes it, if dead.



(a) Expired parameter value triggers pruning of n_2 branch

(b) Expiration of n_2 node triggers validity check of monitor

Figure 3.14.: Pruning of a branch in the indexing tree and monitor validity checking

The monitor is dead if it contains no set of alive bindings so that an accepting state is reachable. These sets of alive bindings, or *aliveness sets*, depend ideally on exact knowledge of the state of the monitor. Due to the paradigm of formalism-independence the monitor is considered as a black-box. The only information we can use is the domain of the monitor instance. The state of the monitor may be any state which can be reached with induction of a parameter set matching this domain. Therefore, Jin and Meredith [20] propose a method to check monitor expiration on basis of the instance domain. The information loss introduced by submitting to formalism-independence for the expiration check is apparent. Because parameter sets can correlate with multiple states, the knowledge derived from runtime is less accurate. This effect increases with a rising number of monitor states in the specification as more states are subsumed under the domain of a specific monitor instance. Consequently, predictions about monitor aliveness must assume the most optimistic internal state regarding success of the trace match. This will be too optimistic in certain cases, causing increased runtime overhead. It is left to future work to explore a selective formalism-dependence for cases where additional information, like the concrete state of the FSM, can be exploited to increase accuracy of expiration predictions.

3.2.3 Representation of Bindings

Parametric instances are represented using arrays. Since the number of parameters is finite, it is possible to represent instances as sequences of parameter values $v \in V$ with length $|\mathcal{X}|$, while the token _ denotes undefined parameter values, or null-values. Because of these unused entries in the sequence, the bindings are called uncompressed bindings.

Definition 26. ***Uncompressed bindings.*** *Let θ be a parametric instance in $[X \rightharpoonup V]$ and $\prec_X$ an order defined over X, then the bindings $\vec{b}_\theta$ denote the sequence $\langle \theta \restriction_{def} (x) \mid x \in X \text{ in order of } \prec_X \} \rangle$ with function $\cdot \restriction_{def} : [X \rightharpoonup V] \to [X \rightharpoonup V]$ defined as*

$$\theta \restriction_{def} = \begin{cases} \theta(x) & \textit{when } \theta(x) \textit{ is defined} \\ \_ & \textit{otherwise} \end{cases}$$

Given the instance $\langle x_1 \mapsto v_1, x_3 \mapsto v_3 \rangle$ with $X = \{x_0, x_1, x_2, x_3, x_4\}$, the uncompressed bindings $\vec{b}_\theta$ are represented by the sequence $\langle \_, v_1, \_, v_3, \_ \rangle$. If an order is defined over $\mathcal{X}$, an index can be calculated for each parameter $x \in X$ which is associated with the position in the bindings.

Definition 27. ***Parameter index.*** *Let $\prec_X$ be an order defined over X, then the parameter index is the function $\textsc{Idx} : \mathcal{X} \to \mathbb{N}_{\geq 0}$ defined as $\textsc{Idx}(x) = |\{x' \mid x' \in X \wedge x' \prec_X x\}|$.*

With a parameter index, the association of the parameter to the parameter value can be reconstructed from the uncompressed binding. Each parameter value v_i has the position i in the sequence with only one $x \in X$ such that with $\textsc{Idx}(x) = i$. This bindings representation is useful for computations since it allows simple and fast access to parameter values. E.g., uncompressed bindings can be used to reference instances which are strictly less informative than the instance represented with the bindings. The bindings can be masked using a parameter set, representing the part of the binding which is selected. The masking can be realized by using the ordering of parameters for indexing the array. Operations over bindings are mostly iterations. An operation f on a sub-instance is just a matter of iterating over a part of the array.

```
for all $x \in \mathcal{X}$ in order of $\prec_X$ do
    $f(\vec{b}[\textsc{Idx}(x)])$
end for
```

Since the representation contains null-values, its memory footprint is not optimal as it does not scale with the number of parameters. Assume, the property to be monitored has a large number of parameters but the number of parameters associated with base events is rather small. This may produce bindings with large numbers of unused spaces. The monitoring algorithm demands that bindings are stored with each instance, since otherwise find-max and join operations can not be performed. To maintain memory efficiency, a second binding representation is used.

Definition 28. ***Compressed bindings.*** *Let θ be a parametric instance in $[X \rightharpoonup V]$, and $\prec_X$ an order defined over X, then the compressed bindings $\bar{b}_\theta$ denote the sequence $\langle \theta(x) \mid x \in X \text{ in order of } \prec_X \wedge \theta(x) \text{ defined} \} \rangle$.*

The instance $\theta = \langle x_1 \mapsto v_1, x_3 \mapsto v_3 \rangle$ with $X = \{x_0, x_1, x_2, x_3, x_4\}$ is represented as compressed bindings with $\bar{b}_\theta = \langle v_1, v_3 \rangle$. Note that the original association of parameter to parameter value is lost in this representation. It has to be maintained separately to be reconstructed. Since the bindings are stored in the data structure with an instance node, the domain of the instance is known. Given instance θ, each $x \in \textsc{Dom}(\theta)$ is associated to exactly one parameter value in $\bar{b}_\theta$, following the order induced by $\prec_X$, e.g., $\textsc{Dom}(\theta) = \{x_1, x_3\}$ and $\bar{b}_\theta = \langle v_1, v_3 \rangle$ from above example.

Incoming events $e\langle\theta\rangle$ with $\{\mathcal{X} \mid \mathcal{X} \in \text{PartEnable}_{\mathcal{G}}^{X}(e) \wedge \mathcal{D}_{\mathcal{E}}(e) \nsubseteq \mathcal{X} \wedge \mathcal{X} \not\subset \mathcal{D}_{\mathcal{E}}(e)\} \neq \emptyset$ enter the join phase. Combinations $\theta \sqcup \theta'$ with a number of compatible binding θ' are created. Using uncompressed and compressed binding representations, this operation can be efficiently computed. Algorithm JoinBindings specifies a method to create a combination of two bindings. It defines the function JoinBindings : $([\langle v_j, ...v_k\rangle] \times [\langle v_l, ...v_m\rangle]) \rightarrow [\langle v_n, ...v_o\rangle] \in v_j, v_k, v_l, v_m, v_n, v_o \in V$ and divides the calculation of the combination into multiple functions. Two of these functions compute patterns which represent instructions for operations on bindings. It is assumed that the incoming instance θ is represented as an uncompressed binding $\vec{b}_\theta$. The join partner θ' is compatible with θ and stored in an instance node. θ' is, therefore, represented as the compressed binding $\bar{b}_{\theta'}$. The result of the combination will be stored with an instance node, therefore, it has to represented as compressed binding.

Dom(θ)	$\vec{b}_\theta$	Dom(θ')	$\bar{b}_{\theta'}$	p_{exp}	p_{copy}	$\bar{b}_{\theta\sqcup\theta'}$
$\{x_1\}$	$\langle \_, v_1, \_, \_, \_\rangle$	$\{x_3\}$	$\langle v_3\rangle$	$\langle 1, -1\rangle$	$\langle 0, 1\rangle$	$\langle v_1, v_3\rangle$
$\{x_3\}$	$\langle \_, \_, \_, v_3, \_\rangle$	$\{x_1\}$	$\langle v_1\rangle$	$\langle -1, 3\rangle$	$\langle 0, 0\rangle$	$\langle v_1, v_3\rangle$
$\{x_0, x_1\}$	$\langle v_0, v_1, \_, \_, \_\rangle$	$\{x_1, x_2\}$	$\langle v_1, v_2\rangle$	$\langle 0, 1, -1\rangle$	$\langle 1, 2\rangle$	$\langle v_0, v_1, v_2\rangle$
$\{x_2, x_4\}$	$\langle \_, \_, v_2, \_, v_4\rangle$	$\{x_0, x_2\}$	$\langle v_0, v_2\rangle$	$\langle -1, 2, 4\rangle$	$\langle 0, 0\rangle$	$\langle v_0, v_2, v_4\rangle$
$\{x_0, x_1, x_4\}$	$\langle v_0, v_1, \_, \_, v_4\rangle$	$\{x_2, x_3, x_4\}$	$\langle v_2, v_3, v_4\rangle$	$\langle 0, 1, -1, -1, 4\rangle$	$\langle 0, 2, 1, 3\rangle$	$\langle v_0, v_1, v_2, v_3, v_4\rangle$
$\{x_0, x_1, x_2\}$	$\langle v_0, v_1, v_2, \_, \_\rangle$	$\{x_2, x_3, x_4\}$	$\langle v_2, v_3, v_4\rangle$	$\langle 0, 1, 2, -1, -1\rangle$	$\langle 1, 3, 2, 4\rangle$	$\langle v_0, v_1, v_2, v_3, v_4\rangle$

Table 3.4.: Expansion and copy patterns

An computation of Join($\vec{b}_\theta, \bar{b}_{\theta'}$) is performed as follows: First, the expansion pattern p_{exp} for $\vec{b}_\theta$ is retrieved and applied to create $\bar{b}_{\theta\sqcup\theta'}$, the sequence representing the yet uncompleted combined binding. $\bar{b}_{\theta\sqcup\theta'}$ is a transformation of $\vec{b}_\theta$ to a compressed binding representation with null-values representing the yet missing parameter values. The term "expansion" signifies that it is converted to the compressed representation, and then expanded, revealing blank spaces expecting parameter values from $\bar{b}_{\theta'}$. These spaces are filled when the copy pattern is applied. The copy pattern p_{copy} represents instructions encoding which parameter value from which position in $\bar{b}_{\theta'}$ has to be copied on which position in $\bar{b}_{\theta\sqcup\theta'}$. Because of two different binding representations, the calculation of the copy pattern may appear complicated at first sight. Table 3.4 shows a number of examples for combinations computed with Join.

Note that the patterns can be precomputed since they only depend on instance domains. The expansion of the uncompressed input bindings has to be performed only once for each set of instances θ' with the same domain, because it depends only on Dom(θ'), not on concrete instances. This allows to process monitor sets more efficiently, since the expanded binding can be copied for each combination to be created with instances having the same domain.

```
JoinBindings
Input: $\text{Idx} : X \rightarrow \mathbb{N}_{\geq 0}$

function Join($\vec{b}_\theta, \bar{b}_{\theta'}$)
    $p_{exp} \leftarrow$ ExpansionPattern(Dom($\theta$), Dom($\theta'$))
    $p_{copy} \leftarrow$ CopyPattern(Dom($\theta$), Dom($\theta'$))
    $\bar{b}_{\theta \sqcup \theta'} \leftarrow \langle ..., \_, ... \rangle$ s.t. Length($\bar{b}_{\theta \sqcup \theta'}$) = Length($p_{exp}$)
    $j \leftarrow 0$
    for all $i \in p_{exp}$ do
        if $i >= 0$ then
            $\bar{b}_{\theta \sqcup \theta'}[j] \leftarrow \vec{b}_\theta[i]$
        end if
        $j \leftarrow j + 1$
    end for
    while $i <$ Length($p_{copy}$) do
        $\bar{b}_{\theta \sqcup \theta'}[p_{copy}[i+1]] \leftarrow \bar{b}_{\theta'}[p_{copy}[i]]$
        $i \leftarrow i + 2$
    end while
    return $\bar{b}_{\theta \sqcup \theta'}$
end function

function ExpansionPattern($\mathcal{X}, \mathcal{X}'$)
    $p = \langle\rangle$
    for all $x \in \mathcal{X} \cup \mathcal{X}'$ ordered by Idx($x$) do
        if $x \in \mathcal{X}'$ then
            $p :: \langle$Idx($x$)$\rangle$
        else
            $p :: \langle -1 \rangle$
        end if
    end for
    return $p$
end function

function CopyPattern($\mathcal{X}, \mathcal{X}'$)
    $p \leftarrow \langle\rangle$
    $i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
    while $i < |\mathcal{X}|$ or $j < |\mathcal{X}'|$ do
        if $i < |\mathcal{X}|$ and $j < |\mathcal{X}'|$ and Idxs($\mathcal{X}$)[$i$] = Idxs($\mathcal{X}'$)[$j$] then
            $i \leftarrow i + 1; j \leftarrow j + 1$
        else if $i < |\mathcal{X}|$ and ($j \geq |\mathcal{X}'|$ or Idxs($\mathcal{X}$)[$i$] < Idxs($\mathcal{X}'$)[$j$]) then
            $i \leftarrow i + 1$
        else
            $p \leftarrow p :: \langle j, i + k \rangle$
            $j \leftarrow j + 1$; $k \leftarrow k + 1$
        end if
    end while
    return $p$
end function

function Idxs($\mathcal{X}$)
    $idxs \leftarrow \langle\rangle$
    for all $x \in \mathcal{X}$ ordered by Idx($x$) do
        $idxs :: \langle$Idx($x$)$\rangle$
    end for
    return $idxs$
end function
```

4 Implementation

The algorithm and data structure described in Chapter 3 have been implemented as a Java library. The library is named prm4j[1] which is an abbreviation of "parametric runtime monitoring for Java". In this chapter we describe the structure of the library and the relationships between the objects.

The library is partitioned into three packages. The `api` package comprises all classes necessary to specify parametric properties, instantiate a parametric monitor and create and pass parametric events. The package `api.fsm` contains a finite-logic specification formalism implemented as FSM. The `spec` package contains interfaces for specification formalisms and provides the base implementation for finite-logic properties. The `indexing` package contains most of the real-time processing part of the algorithm described in Section 3.1.2. Classes in `indexing.model` implement the startup-time processing part. The other packages in `indexing` comprise components of the indexing tree. Cyclic dependencies in the `indexing` package are due to back references needed for the manual reference management and garbage collection.

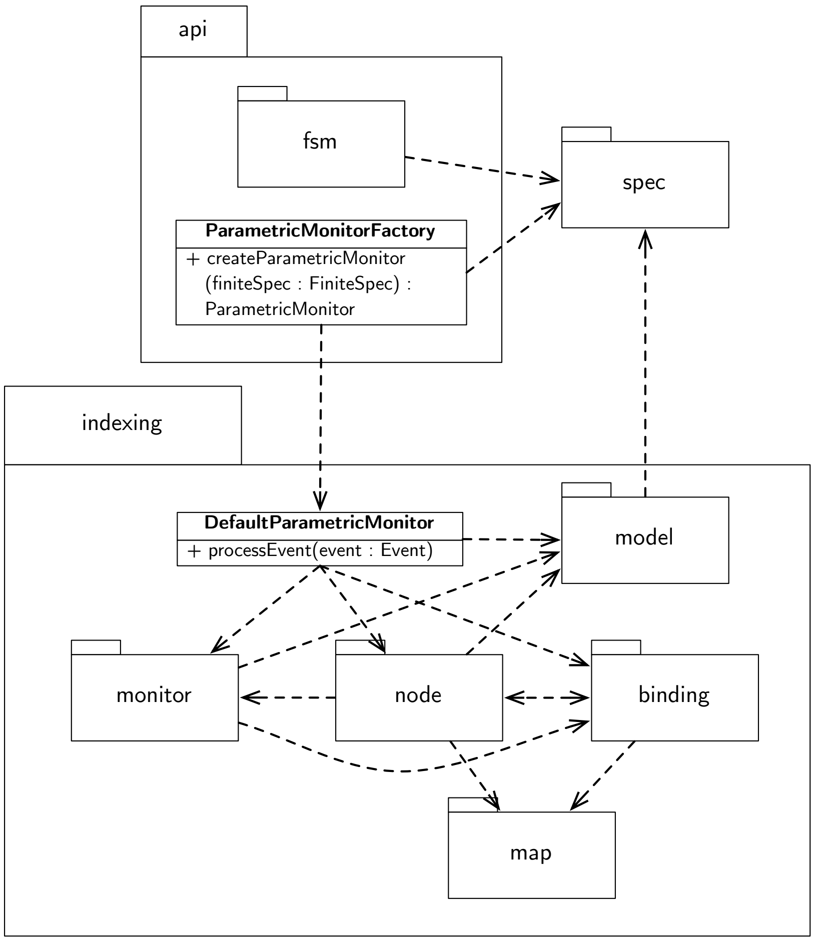

api
fsm
spec
ParametricMonitorFactory
+ createParametricMonitor (finiteSpec : FiniteSpec) : ParametricMonitor
indexing
DefaultParametricMonitor
+ processEvent(event : Event)
model
monitor
node
binding
map

Figure 4.1.: Package structure

Notation: Class diagrams show subsets of the API and fields depending on context. Associations with missing multiplicities represent 1 to 1 associations. Compositions (black diamond) represent final references or immutable references in a reasonable time frame. Aggregations (white diamond) represent references which are created and/or mutated during runtime or by user interaction. Generics are annotated using Java syntax.

[1] Source code of prm4j: `https://github.com/parzonka/prm4j`

4.1 api Package

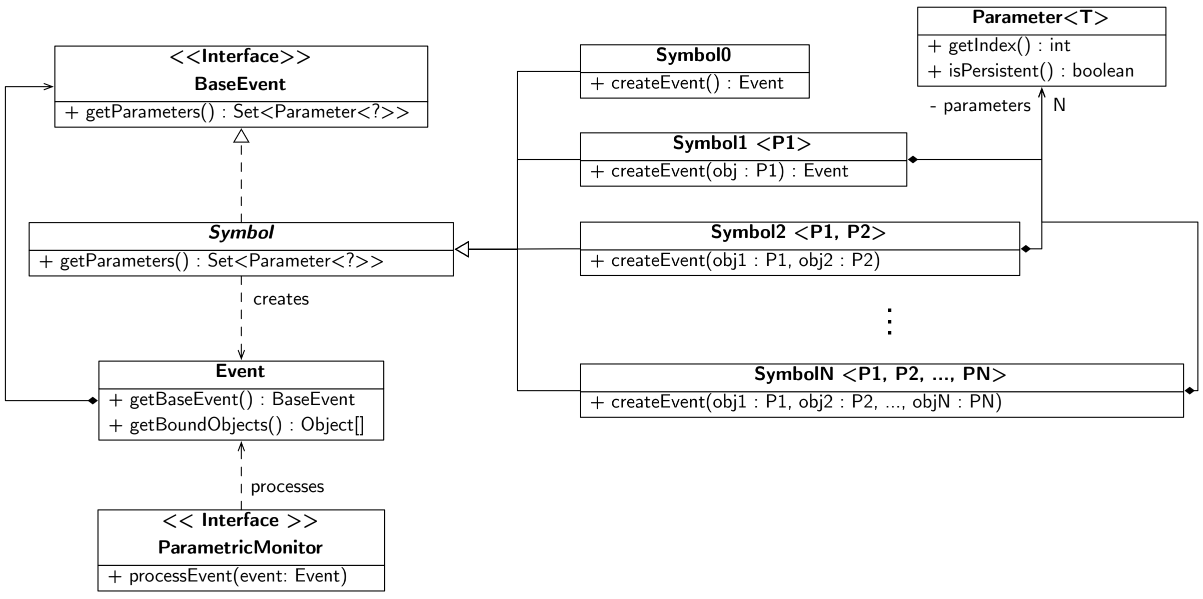
This package provides the API to create a `ParametricMonitor` for the monitoring of a `ParametricProperty`. An implementation for the specification of properties using finite-state machines is introduced.

4.1.1 Parameters, Events and Symbols

A `Parameter` represents the parameter defined in Definition 3. The `Parameter` is a generic class which is instantiated providing the type of the object to be bound. Hence, the `Parameter` encapsulates the parameter value type so that it can be used in the `MatchHandler`.

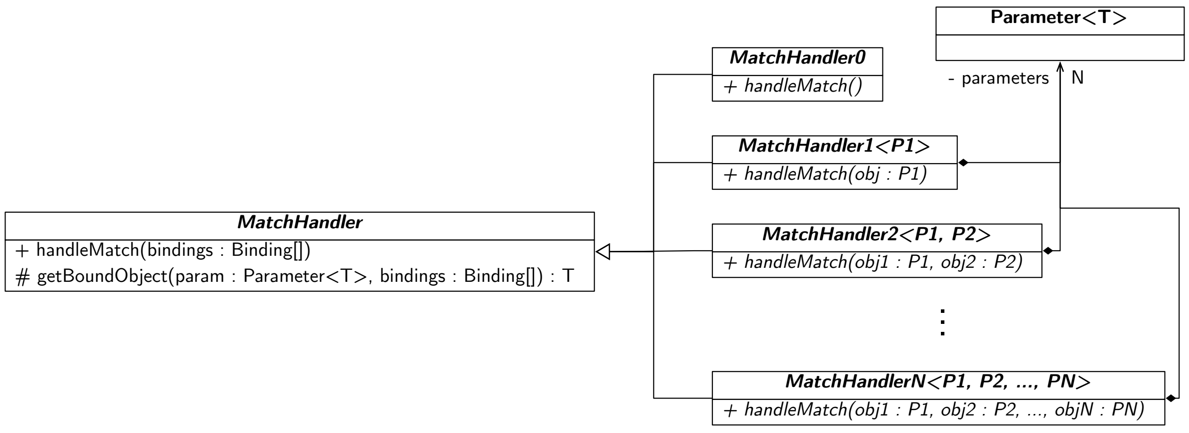
A `BaseEvent` models the base event defined in Definition 1. It stores the associated set of `Parameters` representing the event definition from Definition 5.

The `Symbol` implements the `BaseEvent` and extends it with a user API. Hence, it can be used as part of specification formalisms, e.g., to configure transitions FSMs. The other main purpose of the `Symbol` is the type safe creation of `Event` objects representing parametric events. Each subtype in {`Symbol0`, `Symbol1`, `Symbol2`,...} has a different arity matching the number of parameters which can be bound in the event definition. The `Symbol` provides methods to create `Events` by passing parameter values. Since the `Symbol` is created depending on a number of `Parameters`, the correct type of the parameter values is expected as they are passed to `createEvent`. Created `Events` are passed to the `ParametricMonitor` for processing.



<<Interface>>
BaseEvent
+ getParameters() : Set<Parameter<?>>
Symbol
+ getParameters() : Set<Parameter<?>>
creates
Event
+ getBaseEvent() : BaseEvent
+ getBoundObjects() : Object[]
processes
<< Interface >>
ParametricMonitor
+ processEvent(event: Event)
Symbol0
+ createEvent() : Event
Symbol1 <P1>
+ createEvent(obj : P1) : Event
Symbol2 <P1, P2>
+ createEvent(obj1 : P1, obj2 : P2)
SymbolN <P1, P2, ..., PN>
+ createEvent(obj1 : P1, obj2 : P2, ..., objN : PN)
Parameter<T>
+ getIndex() : int
+ isPersistent() : boolean
- parameters
N

Figure 4.2.: `Events` created by `Symbols` are passed to the `ParametricMonitor`



MatchHandler
+ handleMatch(bindings : Binding[])
# getBoundObject(param : Parameter<T>, bindings : Binding[]) : T
MatchHandler0
+ handleMatch()
MatchHandler1<P1>
+ handleMatch(obj : P1)
MatchHandler2<P1, P2>
+ handleMatch(obj1 : P1, obj2 : P2)
MatchHandlerN<P1, P2, ..., PN>
+ handleMatch(obj1 : P1, obj2 : P2, ..., objN : PN)
Parameter<T>
- parameters
N

Figure 4.3.: The `MatchHandler` type hierarchy

4.1.2 MatchHandler

A `MatchHandler` is an abstraction encapsulating code which is executed on a reported match. The user extends an subtype of desired arity, e.g., MatchHandler0, MatchHandler1, MatchHandler2, etc. and implements the abstract method `handleMatch`. The arity specifies the number of parameter values the `MatchHandler` will use in the match handling code. This code can reference objects belonging to the target program. The parameter values are typed, so the arity also corresponds to the number of type parameters. Figure 4.3 depicts the type hierarchy. Note that the arity symbolized by *N* denotes the highest arity provided in the library.

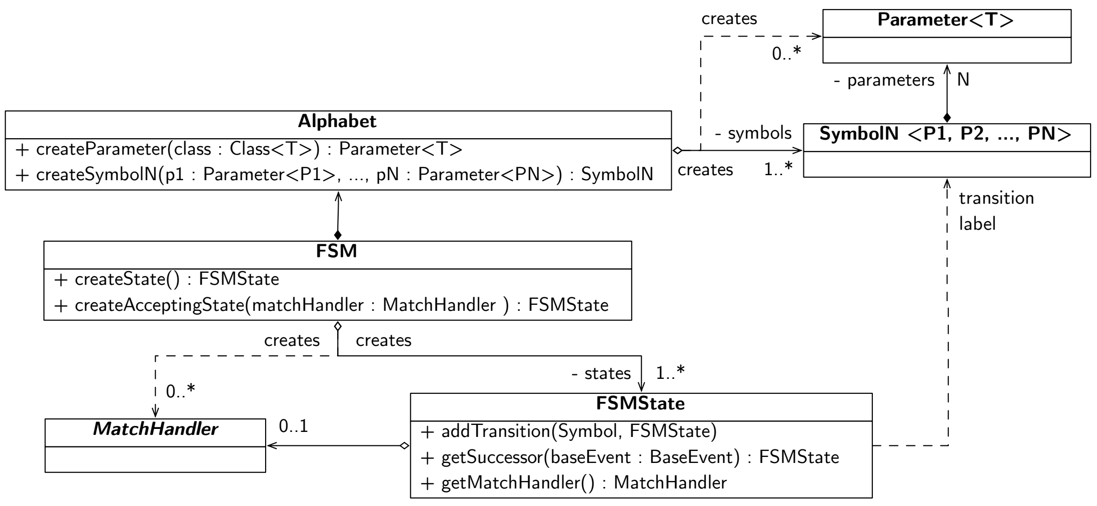
Listing 4.1 shows a custom `MatchHandler2` implementing a match handling method with signature `handleMatch(P1 obj1, P2 obj2, Object auxiliaryData)` with `P1`, `P2` type parameters. The types `Collection` and `Iterator` are provided.

```
class CustomMatchHandler extends MatchHandler2<Collection, Iterator> {

  // initilialization parameters determine which parameter values shall be handled
  public CustomMatchHandler(Parameter<Collection> param1, Parameter<Iterator> param2) {
      super(param1, param2);
  }

  @Override
  public void handleMatch(Collection obj1, Iterator obj2, Object auxiliaryData) {
      System.out.println("Iterator " + obj2 + " was used after collection " + obj1 + "was modified!")
          ;
  }
```

Listing 4.1: Example of a custom `MatchHandler` with arity 2

4.1.3 FSM and Alphabet

Finite-state machines are specified using the `FSM` and `Alphabet` classes as depicted in Figure 4.5. The `Alphabet` creates `Parameters` and `Symbols` which are used in the `FSM`. The `FSM` creates `FSMStates` and connects `FSMStates` with transitions labeled with `Symbols`. `FSMStates` can be created as accepting states providing an optional `MatchHandler`. If an accepting state is reached during monitoring, the parametric monitor will execute the match handling code in the `MatchHandler`.



Parameter<T>
creates
0..*
- parameters
N
Alphabet
+ createParameter(class : Class<T>) : Parameter<T>
+ createSymbolN(p1 : Parameter<P1>, ..., pN : Parameter<PN>) : SymbolN
- symbols
creates
1..*
SymbolN <P1, P2, ..., PN>
transition
label
FSM
+ createState() : FSMState
+ createAcceptingState(matchHandler : MatchHandler) : FSMState
creates
creates
- states
1..*
0..*
MatchHandler
0..1
FSMState
+ addTransition(Symbol, FSMState)
+ getSuccessor(baseEvent : BaseEvent) : FSMState
+ getMatchHandler() : MatchHandler

Figure 4.4.: Classes related to the FSM definition

4.1.4 Usage Example

Listing 4.2 shows a complete monitoring aspect in AspectJ. It encodes the UnsafeIterator property as finite-state machine. This property matches a case where a `Collection` that is in the process of iteration by an `Iterator` is modified and the iteration continues. Event definitions are specified as point cuts. The advice for each point cut creates `Events` based on `Symbols` and passes them to the `ParametricMonitor`. The `CustomMatchHandler` specified in Listing 4.1 is used as `MatchHandler`.

```java
import java.util.Collection;
import java.util.Iterator;

import prm4j.api.Alphabet;
import prm4j.api.MatchHandler;
import prm4j.api.Parameter;
import prm4j.api.ParametricMonitor;
import prm4j.api.ParametricMonitorFactory;
import prm4j.api.Symbol1;
import prm4j.api.Symbol2;
import prm4j.api.fsm.FSM;
import prm4j.api.fsm.FSMSpec;
import prm4j.api.fsm.FSMState;

public aspect UnsafeIterator {

    ParametricMonitor pm;
    Alphabet alphabet = new Alphabet();

    // parameters
    Parameter<Collection> c = alphabet.createParameter(Collection.class);
    Parameter<Iterator> i = alphabet.createParameter(Iterator.class);

    // symbols
    Symbol2<Collection, Iterator> createIter = alphabet.createSymbol2(c, i);
    Symbol1<Collection> updateColl = alphabet.createSymbol1(c);
    Symbol1<Iterator> useIter = alphabet.createSymbol1(i);

    // match handler
    MatchHandler matchHandler = new CustomMatchHandler(c,i);

    FSM fsm = new FSM(alphabet);

    public UnsafeIterator() {

        // fsm states
        FSMState initial = fsm.createInitialState();
        FSMState s1 = fsm.createState();
        FSMState s2 = fsm.createState();
        FSMState error = fsm.createAcceptingState(matchHandler);

        // fsm transitions
        initial.addTransition(updateColl, initial);
        initial.addTransition(createIter, s1);
        s1.addTransition(useIter, s1);
        s1.addTransition(updateColl, s2);
        s2.addTransition(updateColl, s2);
        s2.addTransition(useIter, error);

        // parametric monitor creation
        pm = ParametricMonitorFactory.createParametricMonitor(new FSMSpec(fsm));
    }

    // event definitions
    after(Collection c) returning (Iterator i) :
            (call(Iterator Collection+.iterator()) && target(c)) {
        pm.processEvent(createIter.createEvent(c, i));
    }
    after(Collection c) :
            ((call(* Collection+.remove*(..)) || call(* Collection+.add*(..))) && target(c)) {
        pm.processEvent(updateColl.createEvent(c));
    }
    before(Iterator i) : (call(* Iterator.next()) && target(i)) {
        pm.processEvent(useIter.createEvent(i));
    }

}
```

Listing 4.2: Definition of a monitoring aspect for UnsafeIterator

4.2 spec Package

The spec package contains interfaces for specification formalisms. Specifications are referenced and processed by classes in the `indexing.model` package.

Specifications are represented by the `Spec` interface. A `Spec` comprises `Parameters` and `BaseEvents`, an initial `MonitorState` and an initialized `Monitor` prototype. The prototype is cloned in the real-time algorithm for each monitor created from the initial state. As enforced by formalism-independence, the `Monitor` does not provide access to its `MonitorState`, hence, initial state and prototype monitor have to be provided separately. A `Spec` is agnostic regarding any number of monitor states so an implementation may use finite or infinite numbers of states. The sub-interface `FiniteSpec` extends `Spec` by providing access to a finite set of all possible monitor states. The `FSMSpec` is an implementation of `FiniteSpec` depending on an instance of `FSM` described in Section 4.1.3.

Parametric properties as defined in Definition 10 are represented by the interface `ParametricProperty`. The interface declares all data which has to be provided by a property to be processed by the `ParametricPropertyProcessor`. `FiniteParametricProperty` is an implementation of `ParametricProperty` requiring a finite set of monitor states. It encapsulates the logic needed to extract creation events and enable sets from finite-state properties as defined in Section 3.1.1.2. Hence, `FiniteParametricProperty` is compatible with future specification formalisms in finite logic provided as extensions, e.g., regular expressions or temporal logic. These finite-logic specifications just need to implement `FiniteSpec` to be used with `FiniteParametricProperty`. Specifications for properties using non-finite logic, e.g., context-free grammars have to implement `ParametricProperty` or a dedicated sub-interface.

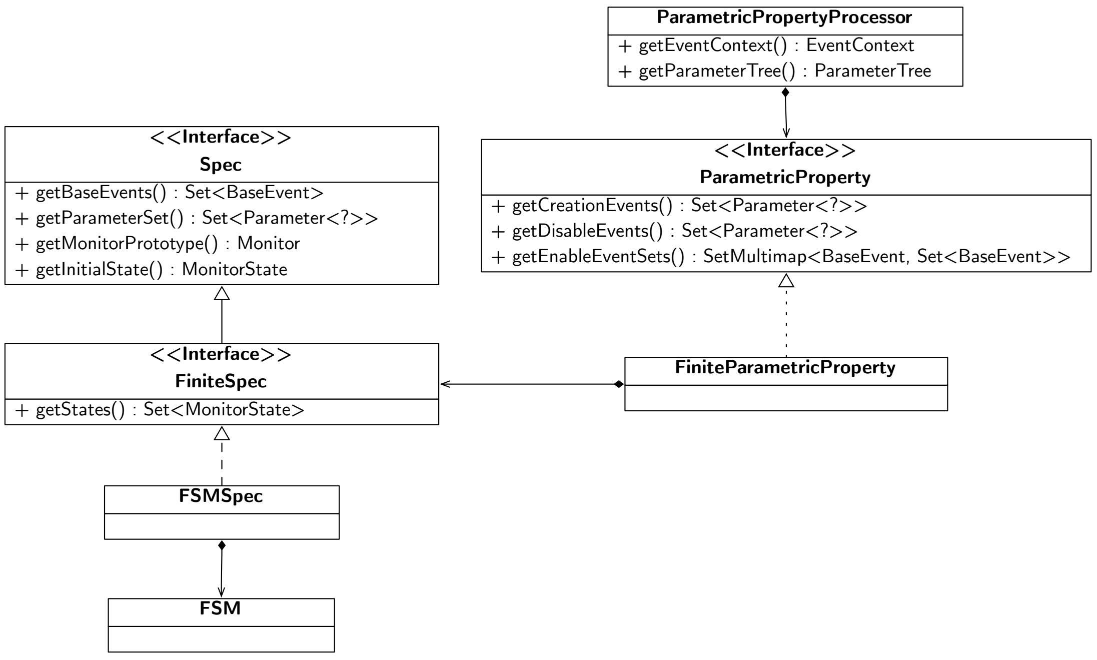
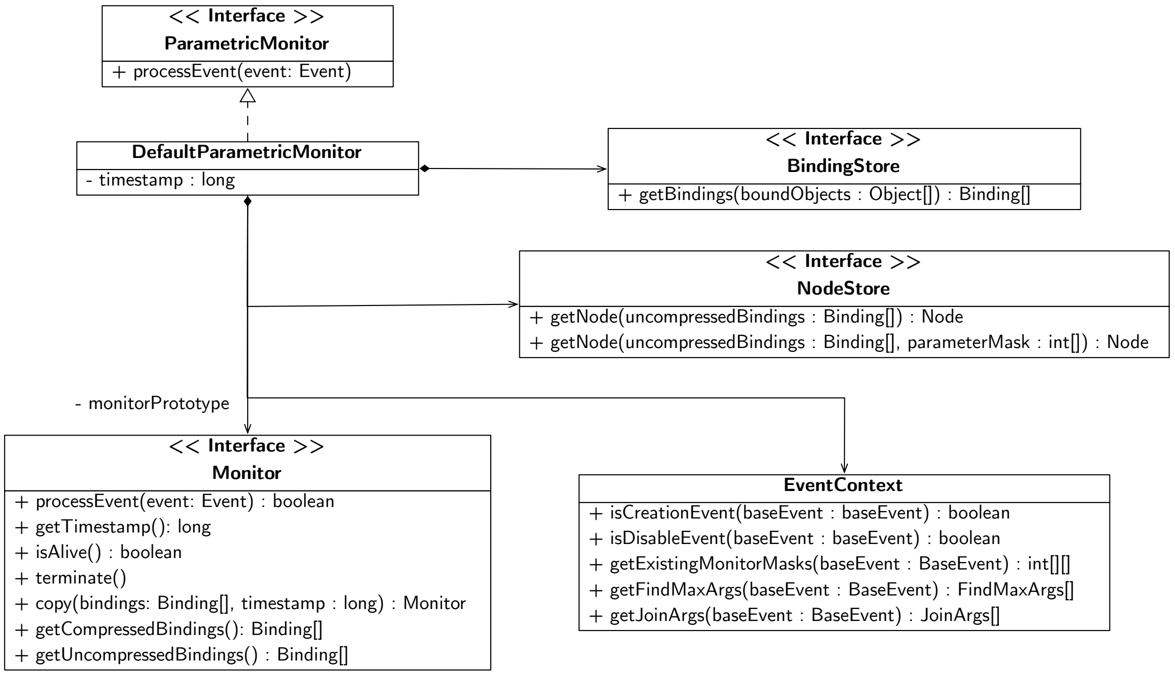

ParametricPropertyProcessor
+ getEventContext() : EventContext
+ getParameterTree() : ParameterTree
<<Interface>>
Spec
+ getBaseEvents() : Set<BaseEvent>
+ getParameterSet() : Set<Parameter<?>>
+ getMonitorPrototype() : Monitor
+ getInitialState() : MonitorState
<<Interface>>
ParametricProperty
+ getCreationEvents() : Set<Parameter<?>>
+ getDisableEvents() : Set<Parameter<?>>
+ getEnableEventSets() : SetMultimap<BaseEvent, Set<BaseEvent>>
<<Interface>>
FiniteSpec
+ getStates() : Set<MonitorState>
FiniteParametricProperty
FSMSpec
FSM

Figure 4.5.: `ParametricProperty` and its finite-logic implementation depending on `FiniteSpec`

4.3 `indexing` Package

The `indexing` package comprises the sub-packages `binding`, `map`, `model`, `monitor` and `node` together with the class `DefaultParametricMonitor`. This class provides the default implementation of the `ParametricMonitor` interface and is the centre of the `indexing` package. It depends on classes from the sub-packages `binding`, `model`, `monitor` and `node`. `DefaultParametricMonitor` implements the real-time part of the parametric trace slicing algorithm described in Chapter 3. `BindingStore` and `NodeStore` form the interface to the underlying data structure. `DefaultParametricMonitor` depends on `EventContext` to encode information to process `Events` in the algorithm. The `EventContext` and a prototype `Monitor` are provided at initialization time of `DefaultParametricMonitor`. The relations of the interfaces and classes are depicted in Figure 4.6.



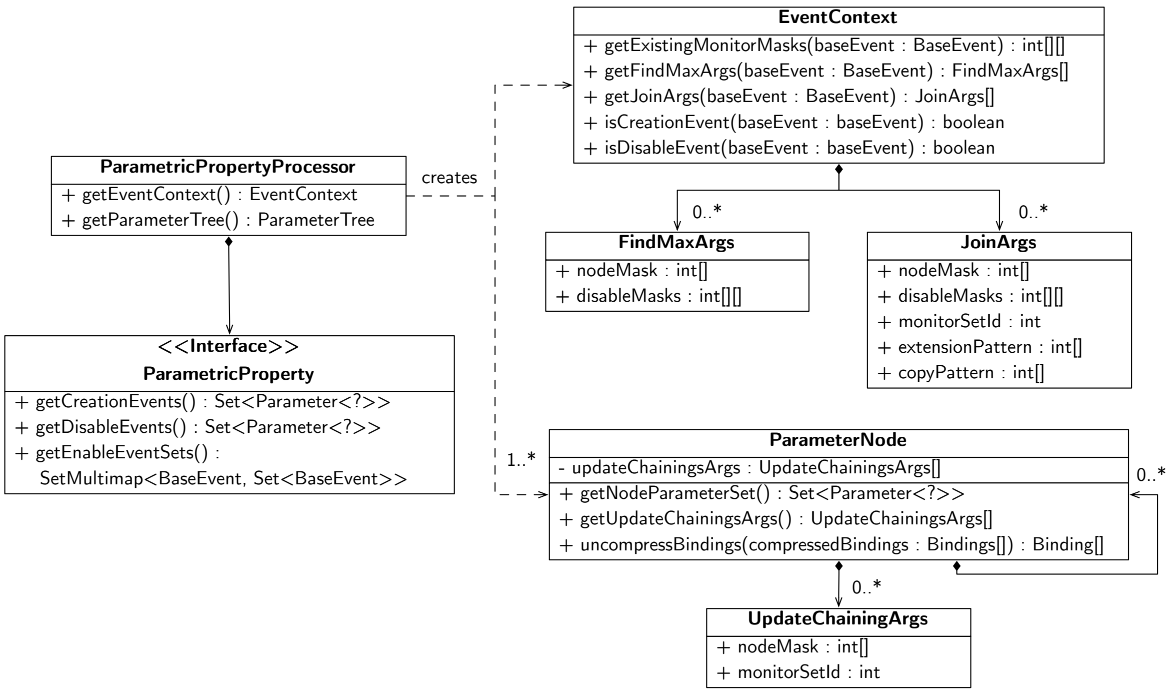
<< Interface >>
ParametricMonitor
+ processEvent(event: Event)
DefaultParametricMonitor
- timestamp : long
<< Interface >>
BindingStore
+ getBindings(boundObjects : Object[]) : Binding[]
<< Interface >>
NodeStore
+ getNode(uncompressedBindings : Binding[]) : Node
+ getNode(uncompressedBindings : Binding[], parameterMask : int[]) : Node
- monitorPrototype
<< Interface >>
Monitor
+ processEvent(event: Event) : boolean
+ getTimestamp(): long
+ isAlive() : boolean
+ terminate()
+ copy(bindings: Binding[], timestamp : long) : Monitor
+ getCompressedBindings(): Binding[]
+ getUncompressedBindings() : Binding[]
EventContext
+ isCreationEvent(baseEvent : baseEvent) : boolean
+ isDisableEvent(baseEvent : baseEvent) : boolean
+ getExistingMonitorMasks(baseEvent : BaseEvent) : int[][]
+ getFindMaxArgs(baseEvent : BaseEvent) : FindMaxArgs[]
+ getJoinArgs(baseEvent : BaseEvent) : JoinArgs[]

Figure 4.6.: Default implementation of `ParametricMonitor` with dependencies

4.3.1 ParametricPropertyProcessor, EventContext and ParameterNodes

The `ParametricPropertyProcessor` encapsulates the most part of the formalism-independent start-up time computation. It is instantiated with a `ParametricProperty` representing a property specified in a given formalism. The `ParametricPropertyProcessor` creates an `EventContext` and a tree of `ParameterNodes`. These two entities can be considered as the real-time model of the consumed parametric property. This model represents the data needed for processing parametric events in the `DefaultParametricMonitor`. The `EventContext` encapsulates the data depending on the concrete `BaseEvent` carried in the current input event. This data is represented by integer arrays and instances of `FindMaxArgs` and `JoinArgs` objects. The `*Args` objects are immutable low-level representations of the argument sequences in algorithm $\mathbb{D}'\langle X\rangle$ as described in Section 3.1.2. All integer arrays except `extensionPattern` and `copyPattern` in `JoinArgs` are parameter masks for compressed binding representations as introduced in Section 3.2.3. E.g., the method `getExistingMonitorMasks` in `EventContext` returns a sequence of parameter masks to select all sub-instances of the input parameter instance which might have already defined monitors. The method `isDisableEvent` tests if a creation event creates a Monitor with a dead state. In this case an memory efficient `DeadMonitor` can be instantiated in the real-time algorithm (see also Section 4.3.8).

The tree of `ParameterNodes` encodes data necessary for event processing which is associated to instance types as defined in Section 3.1.2. It conceptually represents the parameter tree introduced in Definition 25, as it encodes the necessary monitor sets and update relations using $\mathcal{U}_c$-edges. A single `ParameterNode` is associated to all instances $\theta \in [X \rightharpoonup V]$ with the same domain. It stores all data which is relevant for all instances of a domain. This topic is discussed in greater detail in Section 4.3.5.



EventContext
+ getExistingMonitorMasks(baseEvent : BaseEvent) : int[][]
+ getFindMaxArgs(baseEvent : BaseEvent) : FindMaxArgs[]
+ getJoinArgs(baseEvent : BaseEvent) : JoinArgs[]
+ isCreationEvent(baseEvent : baseEvent) : boolean
+ isDisableEvent(baseEvent : baseEvent) : boolean
ParametricPropertyProcessor
+ getEventContext() : EventContext
+ getParameterTree() : ParameterTree
creates
0..*
0..*
FindMaxArgs
+ nodeMask : int[]
+ disableMasks : int[][]
JoinArgs
+ nodeMask : int[]
+ disableMasks : int[][]
+ monitorSetId : int
+ extensionPattern : int[]
+ copyPattern : int[]
<<Interface>>
ParametricProperty
+ getCreationEvents() : Set<Parameter<?>>
+ getDisableEvents() : Set<Parameter<?>>
+ getEnableEventSets() :
SetMultimap<BaseEvent, Set<BaseEvent>>
1..*
ParameterNode
- updateChainingsArgs : UpdateChainingsArgs[]
+ getNodeParameterSet() : Set<Parameter<?>>
+ getUpdateChainingsArgs() : UpdateChainingsArgs[]
+ uncompressBindings(compressedBindings : Bindings[]) : Binding[]
0..*
0..*
UpdateChainingArgs
+ nodeMask : int[]
+ monitorSetId : int

Figure 4.7.: `ParametricPropertyProcessor` creates an `EventContext` and a tree of `ParameterNodes`

4.3.2 Node, NodeRef, Monitor and Bindings

The `Node` represents a node in the indexing tree. `Nodes` are instantiated with a `NodeRef`. A `Monitor` represents a non-parametric monitor as introduced in Definition 11. The `NodeRef` holds a weak reference to a `Node` and a strong reference to a `Monitor`.

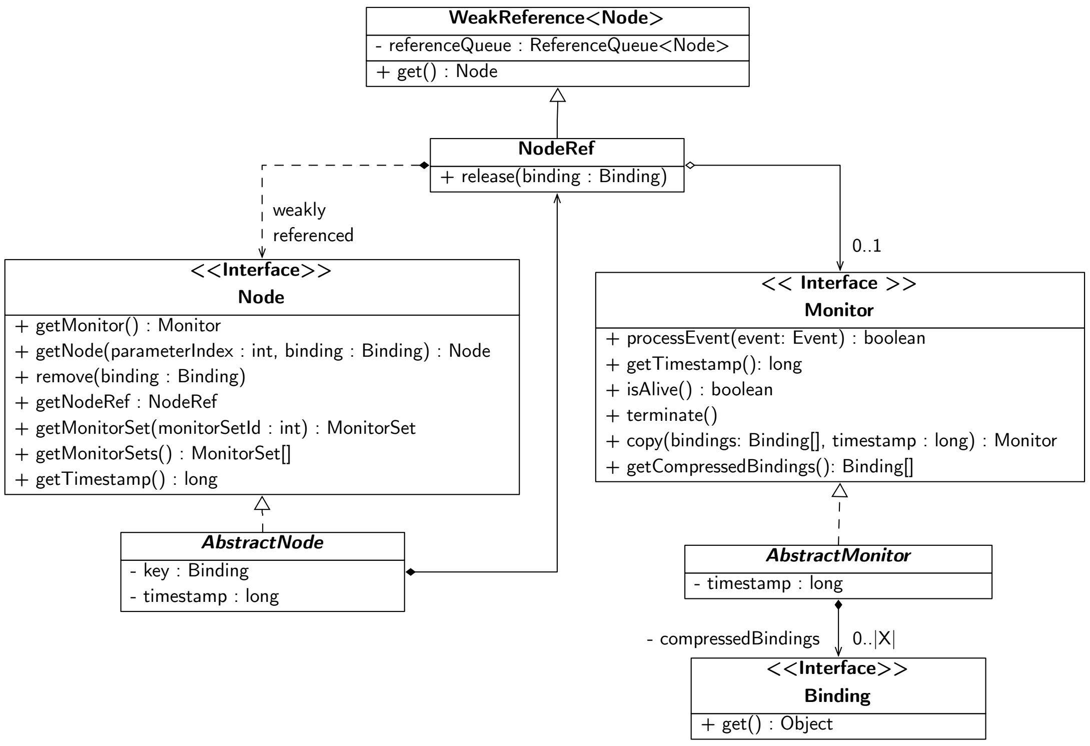
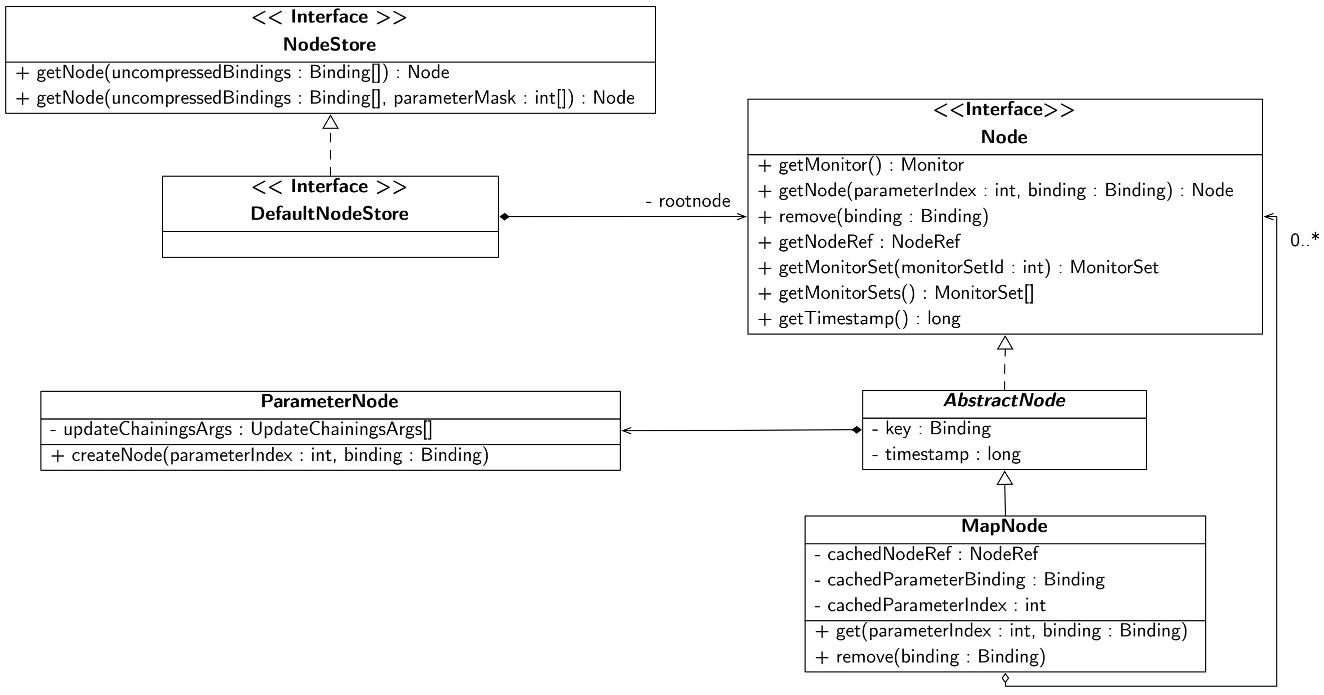


Figure 4.8.: A `NodeRef` with its `Node` and an optional `Monitor`

It can be envisioned as a “bridge” between the Node and Monitor. Not all Nodes have an associated Monitor, e.g., Nodes which do not fully instantiate a parametric instance in the indexing tree. The Binding is an interface representing a referenced Java object in the target program. It is introduced in Section 3.2.1 as an efficient representation of the parameter value defined in Definition 3. Monitors are instantiated with an array of bindings representing the parametric instance.

4.3.3 NodeStore and Nodes

The DefaultNodeStore provides the interface to the indexing tree. The tree is a composition of Nodes with different implementations which are described in Section 4.3.5. The DefaultNodeStore encapsulates the root node of the tree and provides logic for tree traversal. Arrays of Bindings in uncompressed representation are passed to the store which retrieves the associated instance node. The method getNode also accepts Bindings with a parameter mask. The parameter mask selects a subset of Bindings to retrieve sub-instances of the given Binding array.



<< Interface >>
NodeStore
+ getNode(uncompressedBindings : Binding[]) : Node
+ getNode(uncompressedBindings : Binding[], parameterMask : int[]) : Node
<< Interface >>
DefaultNodeStore
- rootnode
<<Interface>>
Node
+ getMonitor() : Monitor
+ getNode(parameterIndex : int, binding : Binding) : Node
+ remove(binding : Binding)
+ getNodeRef : NodeRef
+ getMonitorSet(monitorSetId : int) : MonitorSet
+ getMonitorSets() : MonitorSet[]
+ getTimestamp() : long
0..*
ParameterNode
- updateChainingsArgs : UpdateChainingsArgs[]
+ createNode(parameterIndex : int, binding : Binding)
AbstractNode
- key : Binding
- timestamp : long
MapNode
- cachedNodeRef : NodeRef
- cachedParameterBinding : Binding
- cachedParameterIndex : int
+ get(parameterIndex : int, binding : Binding)
+ remove(binding : Binding)

Figure 4.9.: The DefaultNodeStore stores the root of the indexing tree

A Node is conceptually associated to a parametric instance. A part of its interface forms a specialized map API which allows to retrieve Nodes for a tuple of parameter index and Binding. The parameter index is necessary because a Binding is agnostic regarding the Parameter is it bound to, therefore, it can be bound to multiple Parameters. Without additional information it could not be differentiated which parametric instance has to be identified. Hence, the key to a successor Node in the indexing tree is modeled as a tuple of parameter index and Binding. The remove method, however, does not need to differentiate different parameter indices. Since the only trigger for key removal is expiration of bindings, all mappings for a Binding need to be removed by the Node implementation.

4.3.4 BindingStore, Bindings and MapNodes

All implementations of Binding are based on java.lang.ref.WeakReferences which realize references to Objects without prolonging their lifetime. The Binding implementations are instantiated using a java.lang.ref.ReferenceQueue to detect expiration of bindings. A Binding expires if the referenced parameter value is no longer strongly reachable.

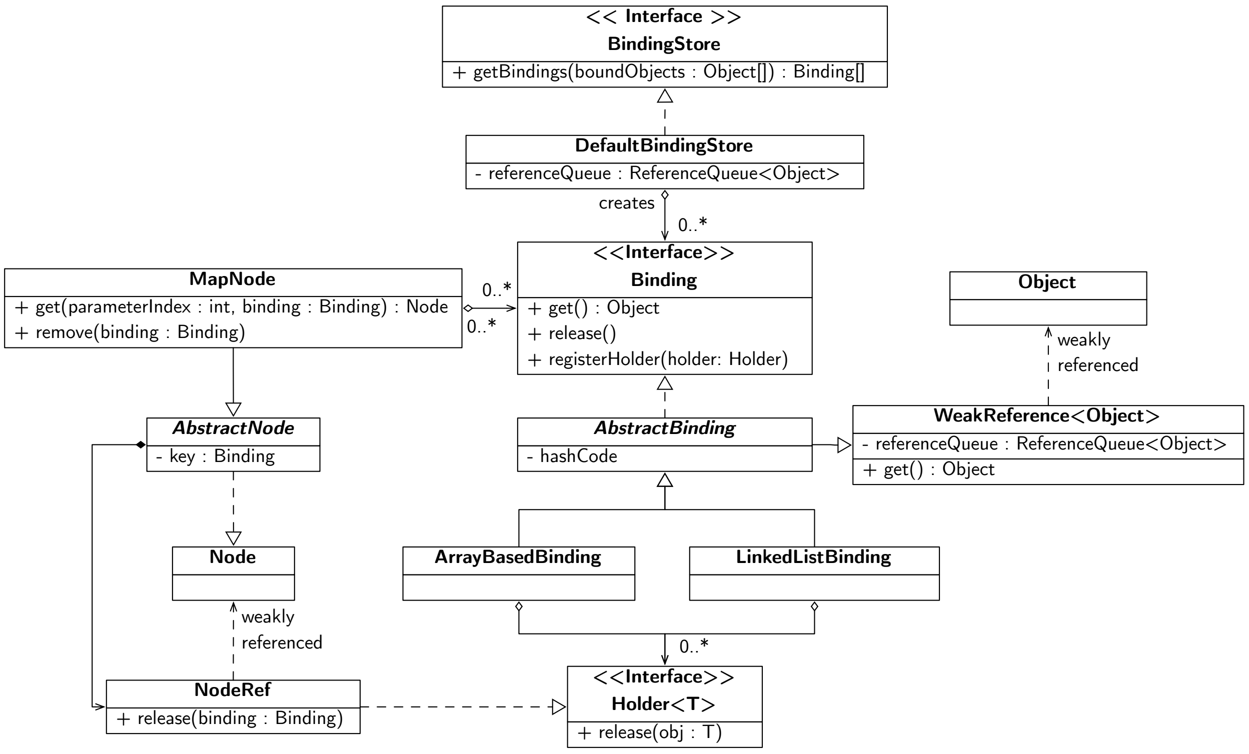


Figure 4.10.: The Binding with its uses as weak parameter value reference and as key in the index tree

Bindings are created and aggregated in an implementation of BindingStore. Its default implementation realizes an identity hash map associating objects in the target application with Bindings. The hash map realizes an association to entries based on the object-identity of the keys using a hash code based on System.identityHashCode. When a Binding does not yet exist in the BindingStore, it is created and initialized to return the same hash code as the bound object. Each subsequent occurrence of the object retrieves the same Binding. Using bindings which store the hash code of the bound object is a technique based on observations by Jin and Meredith [21], where the multiple calculation of the identity hash code was recognized as having a huge performance impact.

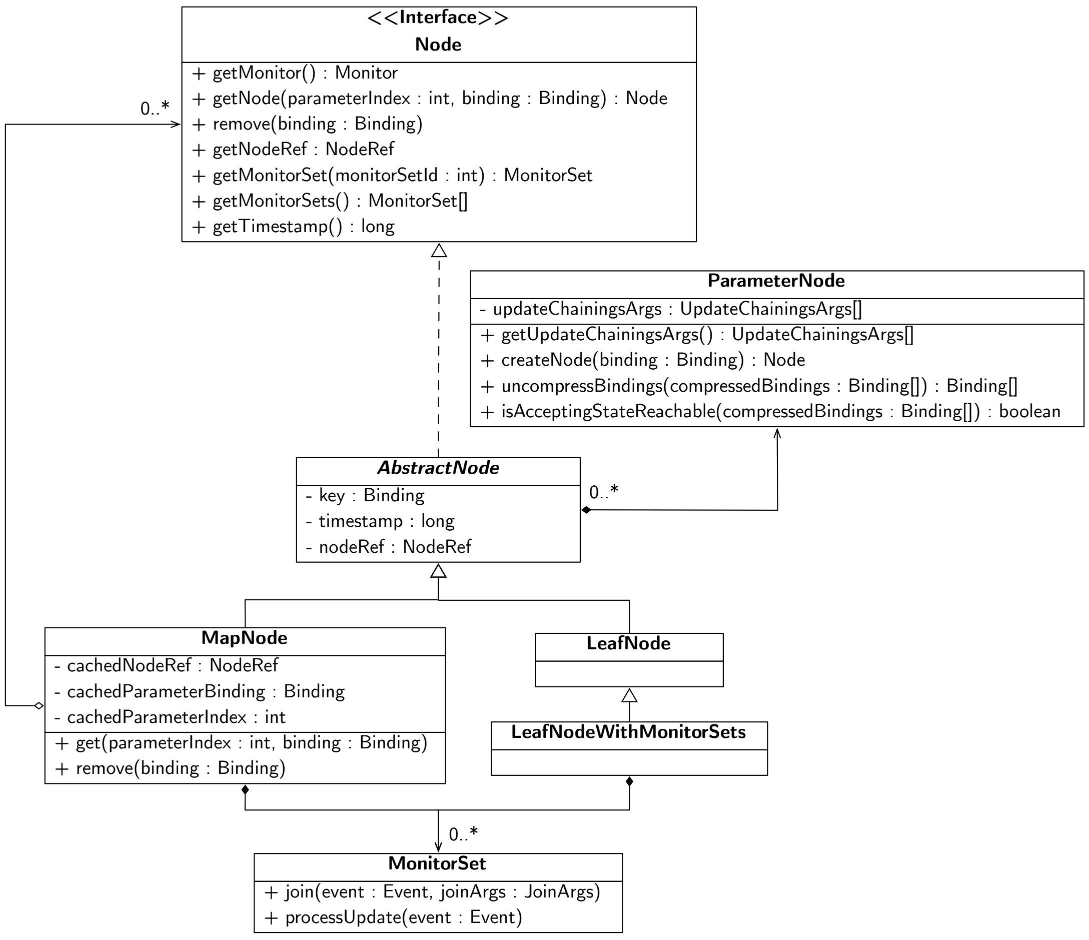
When an Event is processed in the DefaultParametricMonitor it contains an array of Java objects in uncompressed representation considered to be bound to parameters. The array is passed to the BindingStore which returns an array of Bindings in uncompressed representation (see also Section 3.2.3). The position in the array signifies the parameter index. Parameters which are not bound are represented by null-values.

Each Binding implements the registerHolder method. When a Binding is used in a MapNode, this method is called to register the NodeRef as a holder of the Binding. It is then stored in the efficient collection of Holders in the concrete Binding implementation. The ArrayBasedBinding stores Holders in self-resizing arrays while the LinkListBinding stores Holders in a linked list. The generic Holder interface declares a release method. It is implemented in the NodeRef so that a Binding is removed from a MapNode associated to the NodeRef when called. When a Binding is expired, it iterates through all its Holders executing their release method. As the NodeRef weakly references the MapNode it can remove the Binding from the node map when the MapNode is still reachable. This prunes the branch associated with the Binding as described in Section 3.2.2.

The ReferenceQueue is polled at regular intervals (approx. every 5000 events) retrieving expired Bindings. Since calling the poll method has to be synchronized, infrequent polling bursts prevent excessive blocking. In a polling burst, all expired Bindings are retrieved, until the ReferenceQueue returns null.

4.3.5 Node

The main implementation of the `Node` is the `MapNode`. The `MapNode` is a non-leaf node in the indexing tree. It implements the map functionality declared in the `Node` interface. The `MapNode` also implements caching of retrieved nodes. A `NodeRef` is used for caching to prevent leaking memory.



<<Interface>>
Node
+ getMonitor() : Monitor
+ getNode(parameterIndex : int, binding : Binding) : Node
+ remove(binding : Binding)
+ getNodeRef : NodeRef
+ getMonitorSet(monitorSetId : int) : MonitorSet
+ getMonitorSets() : MonitorSet[]
+ getTimestamp() : long
0..*
ParameterNode
- updateChainingsArgs : UpdateChainingsArgs[]
+ getUpdateChainingsArgs() : UpdateChainingsArgs[]
+ createNode(binding : Binding) : Node
+ uncompressBindings(compressedBindings : Binding[]) : Binding[]
+ isAcceptingStateReachable(compressedBindings : Binding[]) : boolean
AbstractNode
- key : Binding
- timestamp : long
- nodeRef : NodeRef
0..*
MapNode
- cachedNodeRef : NodeRef
- cachedParameterBinding : Binding
- cachedParameterIndex : int
+ get(parameterIndex : int, binding : Binding)
+ remove(binding : Binding)
LeafNode
LeafNodeWithMonitorSets
0..*
MonitorSet
+ join(event : Event, joinArgs : JoinArgs)
+ processUpdate(event : Event)

Figure 4.11.: Implementations of `Node` with `ParameterNode` and `MonitorSets`

The `LeafNode` is a memory efficient `Node` providing only a dummy implementation of the node map API. This design decision is a result of a strategy which targets to minimize the number of objects created and maintained in memory. As reported by [19], excessive numbers of even small objects have an impact on memory consumption and garbage collection overhead. Because it is expected that many `Nodes` may be created during monitoring, the map functionality of `Node` is implemented directly in the subclass. We, therefore, favor inheritance over composition which is usually not recommended [15, 5]. An architecturally cleaner solution would be the composition of a base node implementation and a dedicated node map.

Note that all `Nodes` hold a reference to a `ParameterNode`. This is a memory efficient solution to store data which is relevant for all `Nodes` representing instances $\theta \in [X \rightharpoonup V]$ with a specific domain $\textsc{Dom}(\theta)$. E.g., the `ParameterNode` implements the logic and stores the data to convert bindings from the compressed to the uncompressed representation. The `ParameterNode` evaluates binding arrays if acceptable states are reachable. It is also a abstract factory for `Nodes`, since it is configured to create the most memory efficient `Node` implementation for a specific instance type.

The `MapNode` and the `LeafNodeWithMonitorSets` are the only nodes with `MonitorSets`. As described in Section 3.1.3.1, `MonitorSets` realize the $\mathcal{U}_c$ function which allows the selection of `Monitors` associated to a specific parameter set. Multiple `MonitorSets` can be referenced, since each `MonitorSet` selects a specific subset of `NodeRefs` with associated `Monitors`.

4.3.6 MonitorSet

A MonitorSet holds transitive references to Monitors via NodeRefs. It implements the logic to perform the join operation and the update operation introduced in Section 2.2.3. During the join phase, an instance carried by an Event is combined with all instances in the MonitorSet when this operation is allowed by the algorithm. The instances in the MonitorSet are represented by arrays of Bindings in compressed representation stored with Monitors.

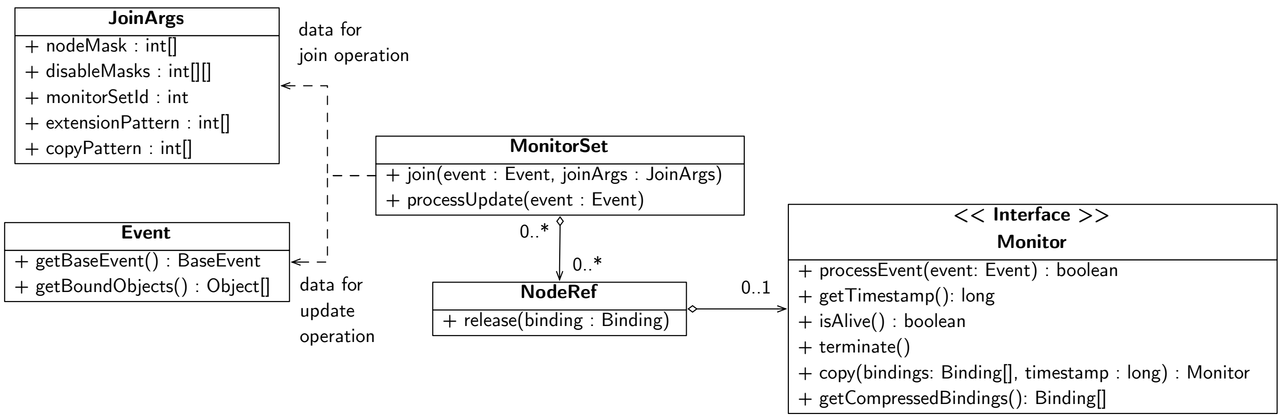

JoinArgs
+ nodeMask : int[]
+ disableMasks : int[][]
+ monitorSetId : int
+ extensionPattern : int[]
+ copyPattern : int[]
data for join operation
MonitorSet
+ join(event : Event, joinArgs : JoinArgs)
+ processUpdate(event : Event)
Event
+ getBaseEvent() : BaseEvent
+ getBoundObjects() : Object[]
data for update operation
0..*
0..*
NodeRef
+ release(binding : Binding)
0..1
<< Interface >>
Monitor
+ processEvent(event: Event) : boolean
+ getTimestamp(): long
+ isAlive() : boolean
+ terminate()
+ copy(bindings: Binding[], timestamp : long) : Monitor
+ getCompressedBindings(): Binding[]

Figure 4.12.: MonitorSet realizes the join operation and update operation

Both operations, join and update, are performed by traversing all Monitors transitively referenced by the MonitorSet. The MonitorSet iterates through its internal array of NodeRefs and checks if a Monitor is referenced. If not, then the NodeRef is removed. If a Monitor is referenced and we are in the join phase, then the array of Bindings is retrieved to create new Monitor with combined bindings. This operation is performed using data from the provided JoinArgs. When in the update phase, then processEvent is called on the Monitor. The Monitor returns a boolean signifying if it is still alive. If it is dead, then the Monitor is removed from the NodeRef and the NodeRef is removed from the MonitorSet.

4.3.7 NodeManager

A ReferenceQueue is associated to the NodeRef determining when a Node ceased to be strongly reachable. In this case, either the Node became unreachable itself or an ancestor was pruned from the indexing tree. The pruning can only happen, if a Binding detects that its referenced object is no longer strongly reachable. The NodeManager holds a reference to the ReferenceQueue which is polled in bursts for NodeRefs with unreachable Nodes.

If a Node is no longer strongly reachable and this is detected by the NodeManager, the monitor of the NodeRef is retrieved, if existing. If the Monitor exists it is considered as *orphaned*, since its associated Node does not exist. On this occasion, it is checked if an accepting state is reachable from the current monitor state using the minimal alive parameter sets retrieved from the ParameterNode. Since a Node can only be garbage collected if it was pruned from the indexing tree, this implies that one of the Bindings associated with Monitor the must have expired. Therefore, it may be possible that some events may not be encountered in the future, thus, a trace slice might never report a match.

4.3.8 Monitor

A Monitor processes Events and updates its internal state accordingly. The StatefulMonitor represents the internal state by maintaining a reference to MonitorState. A MonitorState returns a successor MonitorState for a BaseEvent and provides a MatchHandler which can be executed when an accepting state is reached. A Monitor holds a reference to the ParameterNode of its Monitor. ParameterNode encapsulates the logic and the data to expand the compressed Bindings representation. This is needed when executing MatchHandlers.

An implementation of MonitorState is provided by FSMState. FSMState is also used to specify properties by finite-state machines, as described in Section 4.1.3. Note that the MonitorState is agnostic of its state space, therefore, the state space can be finite or infinite. Hence, the StatefulMonitor should be be compatible with many specification formalisms. The method isAlive returns true if the Monitor may still reach an accepting state.

The DeadMonitor is a memory efficient dummy implementation of Monitor which models initially dead monitors.

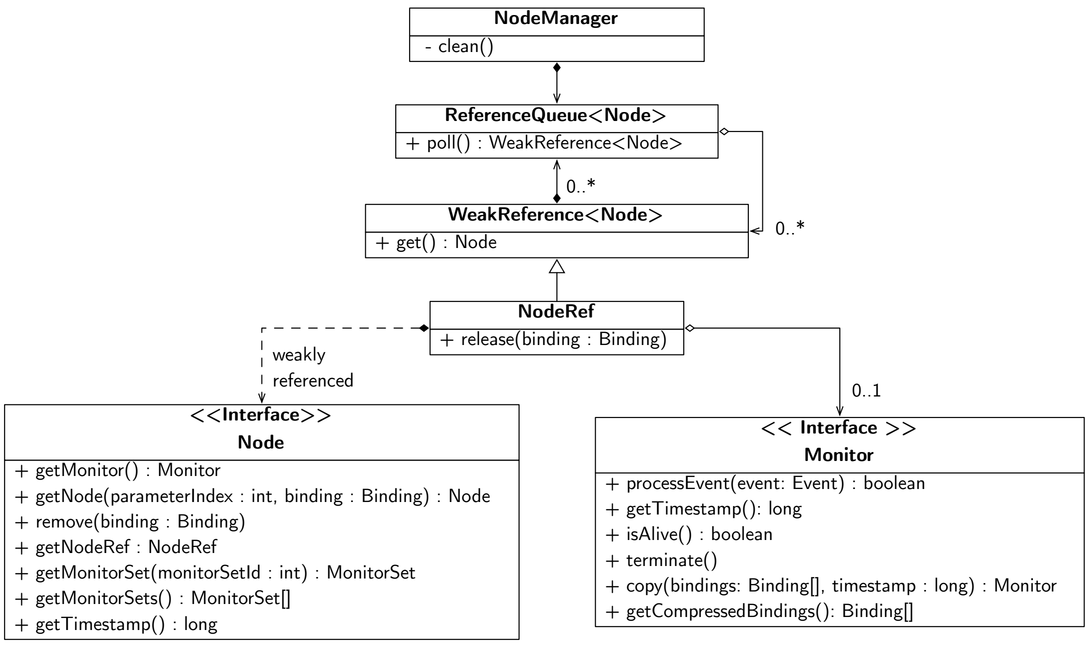

NodeManager
- clean()
ReferenceQueue<Node>
+ poll() : WeakReference<Node>
0..*
WeakReference<Node>
+ get() : Node
0..*
NodeRef
+ release(binding : Binding)
weakly referenced
0..1
<<Interface>>
Node
+ getMonitor() : Monitor
+ getNode(parameterIndex : int, binding : Binding) : Node
+ remove(binding : Binding)
+ getNodeRef : NodeRef
+ getMonitorSet(monitorSetId : int) : MonitorSet
+ getMonitorSets() : MonitorSet[]
+ getTimestamp() : long
<< Interface >>
Monitor
+ processEvent(event: Event) : boolean
+ getTimestamp(): long
+ isAlive() : boolean
+ terminate()
+ copy(bindings: Binding[], timestamp : long) : Monitor
+ getCompressedBindings(): Binding[]

Figure 4.13.: `NodeManager` detects collected `Nodes` and checks aliveness of `Monitors`

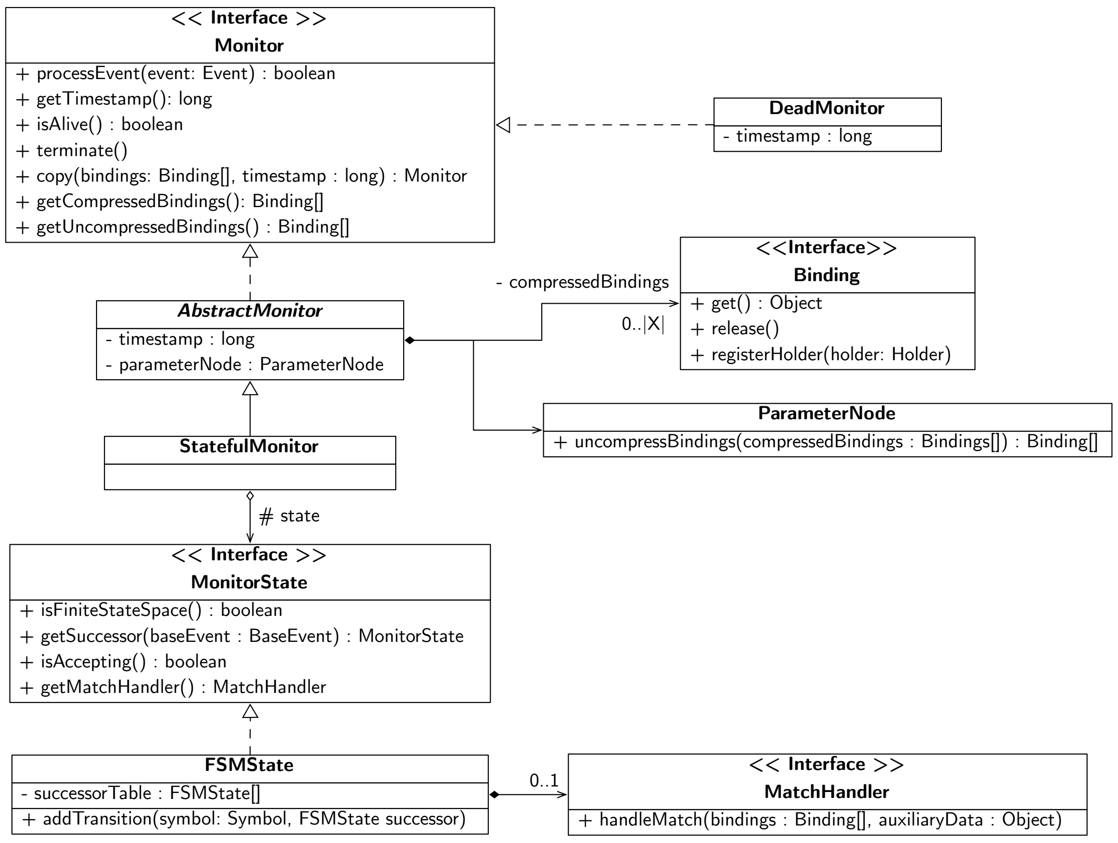

<< Interface >>
Monitor
+ processEvent(event: Event) : boolean
+ getTimestamp(): long
+ isAlive() : boolean
+ terminate()
+ copy(bindings: Binding[], timestamp : long) : Monitor
+ getCompressedBindings(): Binding[]
+ getUncompressedBindings() : Binding[]
DeadMonitor
- timestamp : long
- compressedBindings
0..|X|
<<Interface>>
Binding
+ get() : Object
+ release()
+ registerHolder(holder: Holder)
AbstractMonitor
- timestamp : long
- parameterNode : ParameterNode
ParameterNode
+ uncompressBindings(compressedBindings : Bindings[]) : Binding[]
StatefulMonitor
# state
<< Interface >>
MonitorState
+ isFiniteStateSpace() : boolean
+ getSuccessor(baseEvent : BaseEvent) : MonitorState
+ isAccepting() : boolean
+ getMatchHandler() : MatchHandler
FSMState
- successorTable : FSMState[]
+ addTransition(symbol: Symbol, FSMState successor)
0..1
<< Interface >>
MatchHandler
+ handleMatch(bindings : Binding[], auxiliaryData : Object)

Figure 4.14.: The `StatefulMonitor` implements `Monitor` relying on the `MonitorState`

5 Evaluation

In this chapter we explore how the performance of the library-based approach to parametric runtime monitoring compares to an efficient solutions relying on code generation. Therefore, we compare the performance of prm4j with the performance of JavaMOP, a highly efficient runtime monitoring framework described in Section 2.1.3. In the first section the experimental setup and methodology is described. The results of the comparison are presented in the second section. All results, including evaluation data for JavaMOP, can be reproduced using the evaluation framework prm4j-eval provided online[1]. For answers to detail questions please refer to the source code.

5.1 Experimental Setup

In this section the experimental setup is presented. We motivate a methodology for statistically rigorous performance evaluation. The approach for measurement of runtime performance and memory consumption in prm4j and JavaMOP is described.

5.1.1 Benchmarks and Instrumentation

Following previous work, we use the DaCapo benchmark suite [3] for evaluation. It is a popular tool initially developed for the performance evaluation of different JVM implementations but also widely used in the runtime-monitoring community. The benchmark suite comprises a test harness and benchmark set of open-source, client-side applications. This set is assembled with a focus on realistic, general purpose Java applications that are widely used. It spans a wide range of application domains and behaviors. GUI applications are excluded, since they have proven to be difficult to benchmark systematically. The default input sizes of the programs are designed in such way that they are processed timely enough to allow a statistically rigorous evaluation on commodity hardware in days and weeks instead of months. All benchmarks in the benchmark suite are listed in Table 5.1.

We use all provided benchmarks for the evaluation except of lusearch since we experienced random exceptions thrown in the lucene code. The benchmarks are used with standard input sizes. Instrumentation of the benchmarks is performed using AspectJ load-time weaving. The `aspectjweaver` is passed to the JVM via the `-javaagent` option and weaves all classes which are not excluded in the configuration. We have excluded all classes which are related to the evaluation and instrumentation. DaCapo is executed with the –no-validation option and a custom `org.dacapo.harness.Callback` for runtime measurements which implements the measurement methodology described in Section 5.1.2.

5.1.2 Methodology

Managed runtime systems like Java are challenging to benchmark because of the high number of factors affecting overall performance compared to compiled programming languages as C. Various sources of non-determinism are introduced into the experimental setup inducing variations of execution time from run to run. One source of non-determinism may be external system effects on the evaluation machine. Other sources are inherent in the Java platform like Just-In-Time (JIT) compilation, thread scheduling, garbage collection and sampling-based optimizations in the JVM. Non-determinism causes random errors which are unbiased but have to be handled appropriately in data analysis.

We, therefore, make use of the evaluation methodology proposed by Georges et al. [16] which is designed as a statistically rigorous performance evaluation methodology suitable for the Java platform. Our intention is to measure *steady-state performance*, not start-up performance, because we are interested in the monitoring performance of a long-running system. The first issue when quantifying steady-state performance is to determine when the steady-state is reached. A typical approach is to run the benchmark multiple times within a single VM invocation, i.e., to perform multiple iterations. After a number of iterations, all JIT compilation should be completed and have no impact on the runtime performance of later iterations. These initial iterations are called *warm-up phase*. The DaCapo benchmark suite features a `-converge` option which lets a single benchmark iterate until the relative standard deviation of the determined runtime measures drops below a user-defined value (default is 3%), marking the end of the warm-up phase. DaCapo then runs the benchmark one more time and reports the measured runtime of this last run. Unfortunately this approach does not solve the problem of non-determinism and allows random errors. As reported by Bodden [8], the last run may perform extraordinarily better or worse then previous runs, just by coincidence.

[1] Evaluation framework for prm4j: `https://github.com/parzonka/prm4j-eval`

Benchmark	Description
avrora	Simulates a number of programs run on a grid of AVR microcontrollers
batik	Produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik
eclipse	Executes some of the (non-gui) JDT performance tests for the Eclipse IDE
fop	Takes an XSL-FO file, parses it and formats it, generating a PDF file
h2	Executes a JDBC bench-like in-memory benchmark, executing a number of transactions against a model of a banking application, replacing the hsqldb benchmark
jython	Interprets the pybench Python benchmark
luindex	Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible
lusearch	Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible
pmd	Analyzes a set of Java classes for a range of source code problems
sunflow	Renders a set of images using ray tracing
tomcat	Runs a set of queries against a Tomcat server retrieving and verifying the resulting webpages
tradebeans	Runs the daytrader benchmark via a Java Bean to a GERONIMO backend with an in-memory h2 database
tradesoap	Runs the daytrader benchmark via SOAP to a GERONIMO backend with an in-memory h2 database
xalan	Transforms XML documents into HTML

Adapted from the DaCapo Project: `http://dacapobench.org/`

Table 5.1.: All benchmarks in the DaCapo [3] benchmark suite version 9.12-bach.

We, therefore, iterate the benchmark until the *coefficient of variation* (CoV) of the last five measurements drops below 0.02[2] or we reach a limit of given number of iterations. The coefficient of variation is calculated as the ratio of standard deviation and mean $c_v = \sigma/\mu$. In our case it is calculated for a window of measurements, so standard deviation and mean apply to the same window. When the iteration is completed, the mean of measurements in this window is reported.

Performing multiple iterations is not enough for a statistically rigorous performance evaluation. The second issue with steady-state performance is that different VM invocations may result in different steady-state performances. The reason for this is that different methods may be optimized at different levels of optimizations across different VM invocations, which has an impact on steady-state performance. We, therefore, need to take measurements after reaching steady-state in multiple VM invocations.

Because it is impossible to predict random errors we have to fall back to a statistical model to describe the overall effect of random errors on the expected results. In each benchmark evaluation, a number of runtime measurements is taken. This measurements can be seen as a subset from an infinite number of possible measurements, i.e., the underlying population. Because we cannot collect all possible measurements, it is not possible to get the actual true mean μ for the population. As a sufficient approximate result a confidence interval for the mean can be found which provides a range of values including the mean with a given probability. The confidence interval $[c_1, c_2]$ is defined such that the probability of μ being between c_1 and c_2 equals $(1-\alpha)$. α is called the significance level and $(1-\alpha)$ is called the confidence level. A confidence level of 99% means that there is a 99% probability that the actual distribution mean of the underlying population is within the confidence interval.

To compare two alternative runtime monitoring solutions, the simplest approach is to determine wether the confidence intervals for the two sets of measurements overlap. If they do overlap, then we cannot conclude that the differences seen in the mean values are not due to random errors. In this case, the differences may be possible due to random fluctuations in the measurements. If the confidence intervals do not overlap, then it is legitimate to conclude that there is no evidence to suggest that there is not a statistically significant difference with the given confidence level. Note that there is still a probability of α that the differences observed in our measurements are due to random errors.

Algorithm 11 shows the method to retrieve the confidence interval for a given benchmark. The concrete values are given in the comments.

[2] A CoV-value of 0.02 is a recommendation of Georges et al. [16]. It is equivalent to a relative standard deviation of 2%.

Confidence interval calculation for a single benchmark

```
Input:   invocations : $\mathbb{N}_{>0}$                                   ▷ number of VM invocations
         maxIterations : $\mathbb{N}_{>0}$                   ▷ number of maximum iterations per invocation
         windowSize : $\{1, ..., n\}$                          ▷ number of measurements relevant for mean
         $\alpha$ : $[0, 1]$                                                       ▷ significance level
Output:  confidenceInterval : $(c_1, c_2) \in \mathbb{N}_{>0} \times \mathbb{N}_{>0}$      ▷ confidence interval of measures

1: for all VM invocations $i$ in $\{0, ..., invocations\}$ do
2:     for all benchmark iterations $j$ in $\{0, ..., maxIterations\}$ do
3:         $\bar{x}_i$ = mean of measurements of last windowSize iterations
4:         if $i \geq w$ and CoV of last windowSize iterations is below 0.02 then
5:             go to 11
6:         end if
7:     end for
8: end for
9: return confidence interval for given confidence level $1 - \alpha$ over all $\bar{x}_i$
```

The number of samples we derive from the VM invocations is relatively small, i.e., it will be smaller than 30. In this case it is advised [16, 25] to calculate the confidence interval using the *Student's t-distribution* with

$$c_1 = \bar{x} - t_{1-\alpha/2;n-1} \frac{s}{\sqrt{n}} \text{ and}$$
$$c_2 = \bar{x} + t_{1-\alpha/2;n-1} \frac{s}{\sqrt{n}},$$

where x is the sample mean, s the sample standard deviation, n the number of samples and $t_{1-\alpha/2;n-1}$ is defined such that a random variable T following the Student's t-distribution with $n-1$ degrees of freedom obeys

$$Pr[T \leq t_{1-\alpha/2;n-1}] = 1 - \alpha/2.$$

5.1.3 Monitored Properties

The monitored properties are taken from an evaluation performed by Jin and Meredith [20] to compare the performance of JavaMOP [14], RV [20] and tracematches [1]. Some patterns have also been used in preceding work [10, 7, 14, 29]. All patterns are described in Table 5.2. The UnsafeMapIterator should be already known from Chapter 2 and 3.

The selected properties can be specified in finite logic, hence, they can be defined in prm4j and JavaMOP. We assume the specified properties to be optimized as far as the semantics are not impaired. As both prm4j and JavaMOP depend on proper specification of creation events, either implicit by information encoded in the pattern or explicit by annotation, they are expected to be defined.

Property	Description
HasNext	Matches the case when `hasNext` is not called before the `next` method of an `Iterator`
UnsafeIterator	Matches a case where a `Collection` that is in the process of iteration by an `Iterator` is modified and the iteration continues
UnsafeMapIterator	Like UnsafeIterator with the addition that the `Collection` is created from the key set or value set of a `Map`
UnsafeSyncCollection	Matches the case where either a `Collection` is unsynchronized and a unsynchronized `Iterator` is created for the `Collection`, or a synchronized `Iterator` is created, but accessed in an unsynchronized manner
UnsafeSyncMap	Like UnsafeSyncCollection with the constraint that the `Collection` must be created from a synchronized `Map`

Table 5.2.: Parametric properties used for evaluation

Matches are only reported and no user provided code is triggered. It is not enforced or expected that all bindings belonging to a trace slice are still strongly reachable when a match is reported.

5.1.4 prm4j Setup

The parametric monitors are defined as AspectJ aspects with dependencies to the prm4j library. These aspects can be woven into the target application using any AspectJ compiler or load-time weaving. As described in Section 5.1.1, load-time weaving is used in the evaluation. Properties are specified using the FSM formalism providing creation events encoded as initial self-loops. The event definitions provided as AspectJ point cuts were taken from generated JavaMOP code for the same properties. This ensures comparability with JavaMOP. All specifications are listed in Appendix A.

prm4j contains a built-in logger for memory consumption and various statistics like the number of created and collected bindings and monitors. This logger is only activated when measuring memory consumption as it creates a slight overhead for runtime performance.

Multiple iterations have consequences for the evaluation process. Since the monitoring aspect persists in memory in between iterations, it has to be reset to release resources and reinitialize its state. Parametric monitors in prm4j are resettable via an API call. The reset event is implemented as an additional join point matching the stop of an iteration in the custom DaCapo callback. The reset of the prm4j components is accompanied by several manual executions of the Java garbage collector to ensure sufficient release of memory. The time for this final garbage collection runs is not measured.

5.1.5 JavaMOP Setup

JavaMOP 3.0.0 (build 232)[3] was used for evaluation. It contains the five selected patterns as example specifications in `.mop` format. HasNext is specified using the FSM plugin, the other four are specified using extended regular expressions (ERE). The specifications are listed in Appendix B.

As can be observed, only UnsafeSyncMap and UnsafeSyncColl have annotated creation events. With missing annotations JavaMOP automatically derives the creation events from FSM and ERE properties. It marks all events $e \in \mathcal{E}$ as creation events, for which $\emptyset \in \textsc{Enable}_{\mathcal{G}}^{\mathcal{E}}(e)$. This behavior has been confirmed by the designers of JavaMOP. With this semantics the specifications are equivalent with the specifications of prm4j.

The aspects were generated from the specification and included in the evaluation framework. They depend on the library `javamoprt` providing additional implementation. Since the JavaMOP aspects are not resettable via their API, we have implemented a reset mechanism by including a point cut matching the stop of a benchmark iteration. On recognized reset, all fields in the aspect are reinitialized and several runs of the Java garbage collector initiated. As this happens after the end of an iteration and before the next iteration starts, the time for reinitialization is not measured. A event counter, a counter for matching traces and memory logger were integrated into the generated code. Since the latter has a slight impact on runtime performance it is only activated when measuring memory consumption. All modifications to the generated source code are annotated and can be reviewed in the evaluation framework.

5.1.6 Hardware Setup

We used a Intel Core 2 Duo / 2 GHz / 3 GB Mac OSX 10.6.8 machine and version 9.12 of the DaCapo (DaCapo 9.12-bach) benchmark suite [3] running Java SE Runtime Environment (1.6.0_37_b06_434_10M3909) with HotSpot 64-Bit Server VM (build 20.12-b01-434, mixed mode). Maximum heap size is set to 2GB.

5.2 Results

In this section we present the results documenting correctness, runtime performance and memory consumption of prm4j in comparison to JavaMOP.

5.2.1 Correctness

Correctness is crucial in the evaluation of runtime monitoring approaches. We augmented the runtime monitors of JavaMOP and prm4j with event and match counters to compare the number of events and matches for each property. The results of our experiments show that the numbers are identical. Small deviations in the counted events and matches were observed due to indeterminism in the benchmarks tomcat, tradebeans and tradesoap. Other benchmarks, including avrora, eclipse, h2 and jython, show stable match counts but varying event counts. The counts of those benchmarks vary naturally across different iterations and invocations. Table 5.3 shows the numbers averaged across 16 JVM invocations with multiple iterations.

[3] JavaMOP 3 homepage: `http://fsl.cs.uiuc.edu/index.php/Special:JavaMOP3`

	HasNext		UnsafeIterator		UnsafeMapIterator		UnsafeSyncCollection		UnsafeSyncMap	
Benchmark	Events	Matches	Events	Matches	Events	Matches	Events	Matches	Events	Matches
avrora	1,506,330	103,521	1,364,872.2	0	1,260,521.7	0	0	0	0	0
batik	48,203	646	124,527	0	53,693	0	55,532	0	0	0
eclipse	10,913.1	0	5,983	0	7,322.9	0	0	0	0	0
fop	1,012,142	214,225	704,570	0	491,124	0	1,044,543	0	1,037,108	0
h2	2,683,515.1	0	1,475,393.5	0	1,754,201.6	0	0	0	0	0
jython	442,836.8	0	715,062.9	0	144,594.2	0	438,397	0	0	0
luindex	365	0	4,393	0	342	0	0	0	0	0
pmd	8,284,729	2,492	6,454,491	0	4,313,219	0	8,838,236	0	8,612,806	0
sunflow	2,655,217	0	1,277,075	0	1,276,942	0	0	0	0	0
tomcat	112,912	5.1	100,950.9	0	141,905.5	0	0	0	0	0
tradebeans	21,825,426.1	52.7	11,090,052.3	0	14,751,907.8	0	0	0	0	0
tradesoap	8,876,483.7	62,596	10,511,245.7	0	8,194,929.7	0	0	0	0	0
xalan	5	0	1,309,540.5	0	125,612	0	0	0	0	0

Table 5.3.: Number of events and matches per benchmark and property in prm4j and JavaMOP

During the work on this thesis, we have examined the generated code of JavaMOP for various parametric properties. We observed a deviation from the original theory which we first suspected to be a legit optimization not found in any publication. In the current implementation, JavaMOP does not store the timestamp $\mathcal{T}_\Theta(\theta)$ for an instance $\theta \in [X \rightharpoonup V]$ in a node in its indexing tree but in all its associated bindings. Recall that $\mathcal{T}_\Theta$ returns the last time an unknown instance was seen while $\mathcal{T}_\Delta$ returns the time the initial monitor for a trace slice was created. A binding encapsulates a parameter value being agnostic of the concrete parameter. The resulting variant of algorithm $\mathbb{D}\langle X\rangle$, therefore, deviates in the approach of assigning timestamps. When we interpret the association of a parameter to a binding as an instance defined for only one mapping, the assignment $\mathcal{T}_\Theta(\theta) \leftarrow t$ in algorithm $\mathbb{D}\langle X\rangle$, Line 7 becomes $\mathcal{T}_\Theta(\theta') \leftarrow t$ for each $\theta' \in \{\theta' \mid \theta' \in [X \rightharpoonup V] \wedge \theta' \sqsubseteq \theta \wedge |\textsc{Dom}(\theta')| = 1\}$. In other words, not the whole instance is assigned with the current timestamp, but all its sub-instances representing a single defined mapping. Consequently, the time check in Line 38 is not performed on all sub-instances of an input instance θ, but on all sub-instances consisting of a single mapping. The function DefineTo beginning in Line 37

```
function DefineTo($\theta, \theta'$)
    for all $\theta'' \sqsubseteq \theta$ s.t. $\theta'' \not\sqsubseteq \theta'$ do
        if $\mathcal{T}_\Theta(\theta'') > \mathcal{T}_\Delta(\theta')$ or $\mathcal{T}_\Delta(\theta'') < \mathcal{T}_\Delta(\theta')$ then
            return
        end if
    end for
    ...
end function
```

is substituted by

```
function DefineTo($\theta, \theta'$)
    for all $\theta'' \sqsubseteq \theta$ s.t. $\theta'' \not\sqsubseteq \theta'$ and $|\textsc{Dom}(\theta'')| = 1$ do
        if $\mathcal{T}_\Theta(\theta'') > \mathcal{T}_\Delta(\theta')$ or $\mathcal{T}_\Delta(\theta'') < \mathcal{T}_\Delta(\theta')$ then
            return
        end if
    end for
    ...
end function.
```

However, this approach has a serious issue. It can be shown, that events from different trace slices can interfere with each other. Assume the following example. $X = \{a, b\}, \mathcal{E} = \{e_1, e_2\}$ with $\mathcal{D}_\mathcal{E} = \{e_1 \mapsto \{a\}, e_2 \mapsto \{a, b\}\}$. Let the property P be specified to match the regular expression e_1e_2. When the variant of algorithm $\mathbb{D}\langle X\rangle$ processes the trace $\tau = e_1\langle a_1\rangle e_2\langle a_2b_1\rangle e_2\langle a_1b_1\rangle$ no match is detected. However, the trace slice $\tau \upharpoonright_{\langle a_1b_1\rangle} = e_1e_2$ is exactly the pattern specified in P. Due to the specification of the property and the definition of a trace slice, this must be incorrect. The reason for this is, that on $e_2\langle a_2b_1\rangle$ an instance timestamp $\mathcal{T}_\Theta(\langle b_1\rangle)$ is assigned. When during join phase for $e_2\langle a_1b_1\rangle$ the monitor state is tried to be copied from $\Delta(\langle a_1\rangle)$, this timestamp is used in the modified time check as described above. The monitor is created from the copied state when $\mathcal{T}_\Theta(\langle b_1\rangle) > \mathcal{T}_\Delta(\langle a_1\rangle)$ evaluates to false. This is not the case, because $\langle b_1\rangle$ appears to be seen after the monitor for $\langle a_1\rangle$ is created. Thus, the method returns without deriving the monitor and updating its state. But this is not correct, since only $\langle a_2b_1\rangle$ has been seen after $\langle a_1\rangle$, not $\langle b_1\rangle$ alone. In fact, the latter is not possible anyway, because there is no event $e \in \mathcal{E}$ with $\mathcal{D}_{\mathcal{E}}(e) = \{b\}$. Hence, the algorithm variant currently implemented in JavaMOP must be considered to be seriously flawed as it can not discern instance *tuples* from sole instances.

This issue could be reproduced by generating a monitor aspect showing the expected incorrect behavior. The interested reader is referred to its specification in the Appendix, Listing B. The issue is confirmed by the designers of JavaMOP. A fix could not yet be provided, therefor we perform the evaluation with this version.

The described issue has effects on the evaluation. The effects on runtime performance of JavaMOP are beneficial since bindings can be retrieved considerably faster than indexing tree nodes. In most cases, the bindings are already retrieved at the time of the time check. The correctness of the monitoring result, however, is most probably not impaired since the properties used in the evaluation do not belong to the class of properties affected by the issue.

5.2.2 Runtime Performance

The runtime performance is measured in percentage of the original runtime. Initially, a baseline was created using the DaCapo benchmarks without any instrumentation. The runtime was measured across multiple JVM invocations. Figure 5.1 depicts the measured runtimes. The error bars denote the confidence interval with a confidence level of 99%, assuming a symmetric interval.

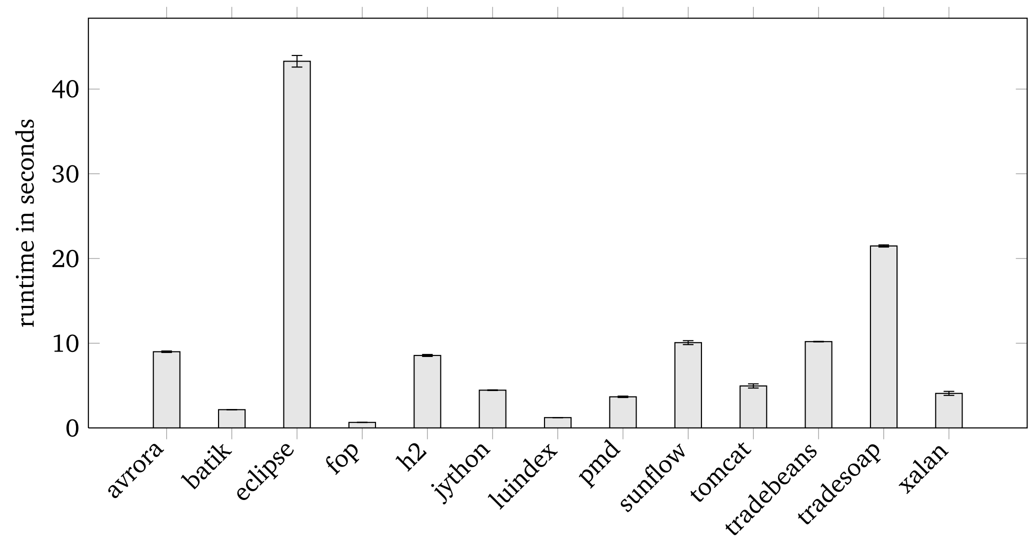
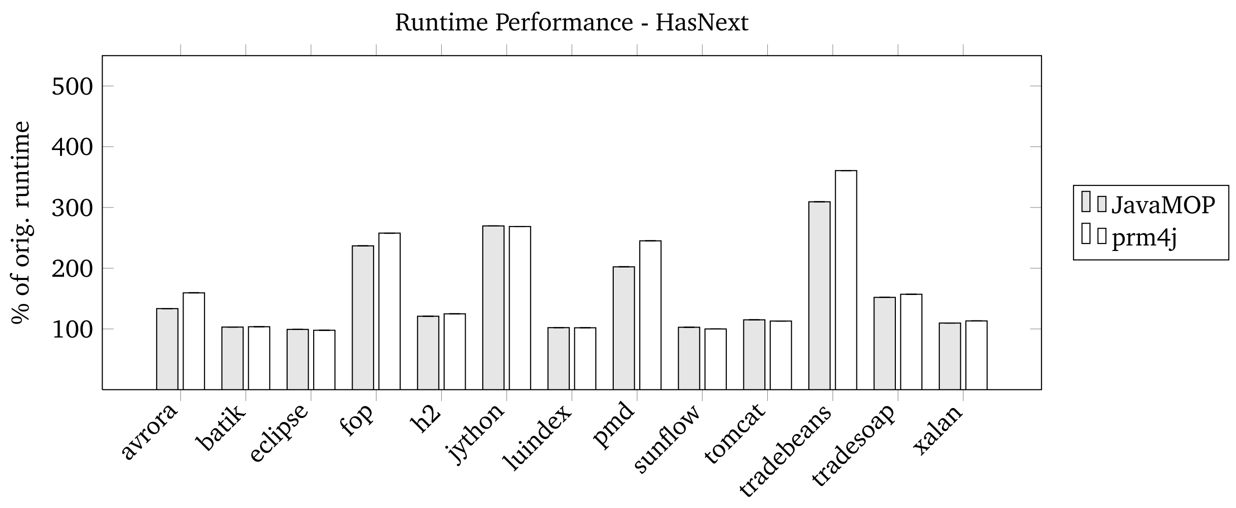


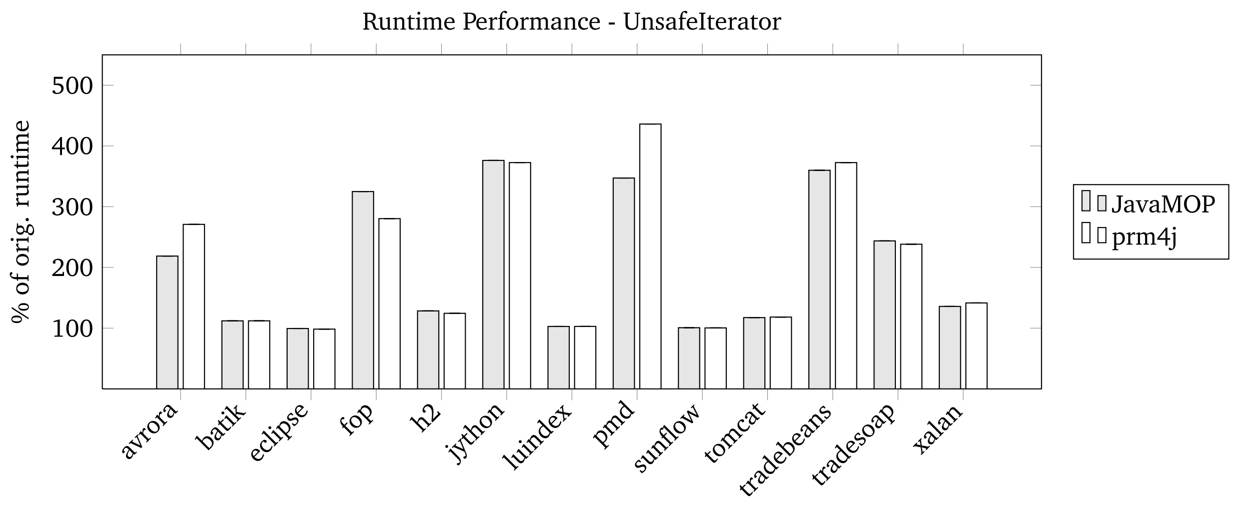
Figure 5.1.: Runtime baseline in seconds with confidence level 99%.

The DaCapo benchmarks were executed on the evaluation system being instrumented with a single runtime monitor at a time. The time was measured using a custom `org.dacapo.harness.Callback` counting milliseconds between start and stop of each iteration. Post-iteration code like monitor resetting and manual calls to the garbage collector have no impact on the time measurements. Measurements were taken as described in the methodology with invocations = 16, maxIterations = 25 and window = 5. Table 5.4 shows a performance comparison in percent of the original runtime. The values in the table represent the mean in the confidence interval. Figures 5.2 and 5.3 visualize the same numbers as bar chart.

Due to the sufficiently large number of invocations the confidence interval width is below 0.01 percent of the original runtime which can be documented in Appendix C. Therefore, the intervals do not overlap and all differences are statistically significant assuming a probability of 0.01 that the differences observed in our measurements are due to random errors.

Considering the property HasNext we can observe JavaMOP is nearly always better with the best result on the avrora benchmark using half was much additional overhead than prm4j. The only benchmark where prm4j is slightly faster is sunflow. For UnsafeIterator prm4j performs worst in avrora with a slight advantage to JavaMOP in fop. The runtime performance for the multitude of these benchmarks can be considered equal. The best performance results in comparison to JavaMOP can be observed when monitoring the UnsafeMapIterator property. prm4j's overhead is lower in all benchmarks except xalan. The best result can be observed with avrora where prm4j's overhead is nearly four times lower than JavaMOP. Notable runtime performance can also be attested with fop, pmd and tradebeans. Considering UnsafeSyncCollection and UnsafeSyncMap, JavaMOP and prm4j deliver comparable runtimes. pmd shows a slightly greater runtime overhead as compared to JavaMOP.





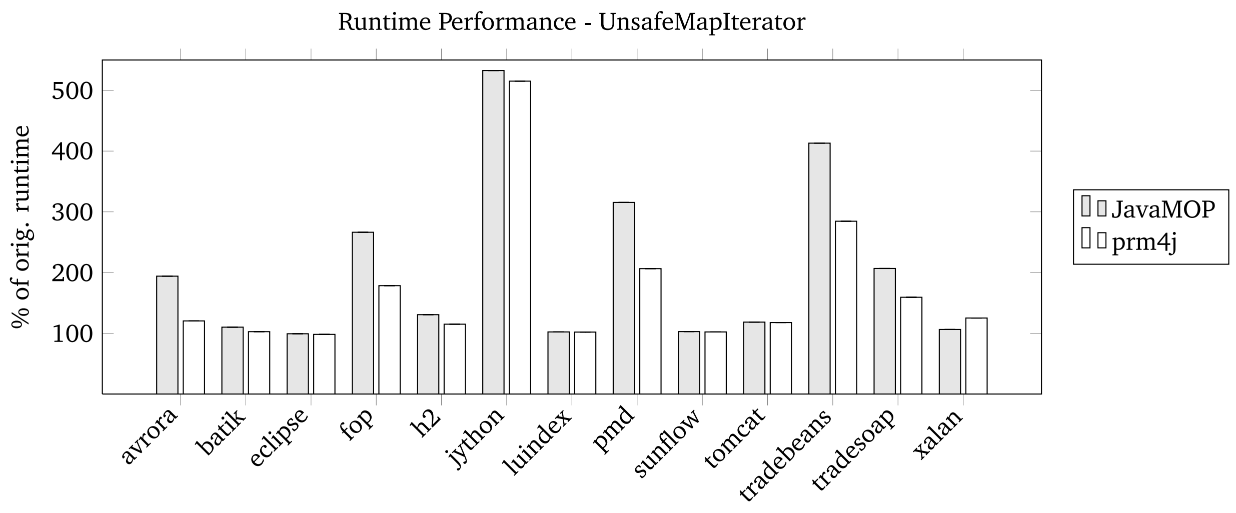
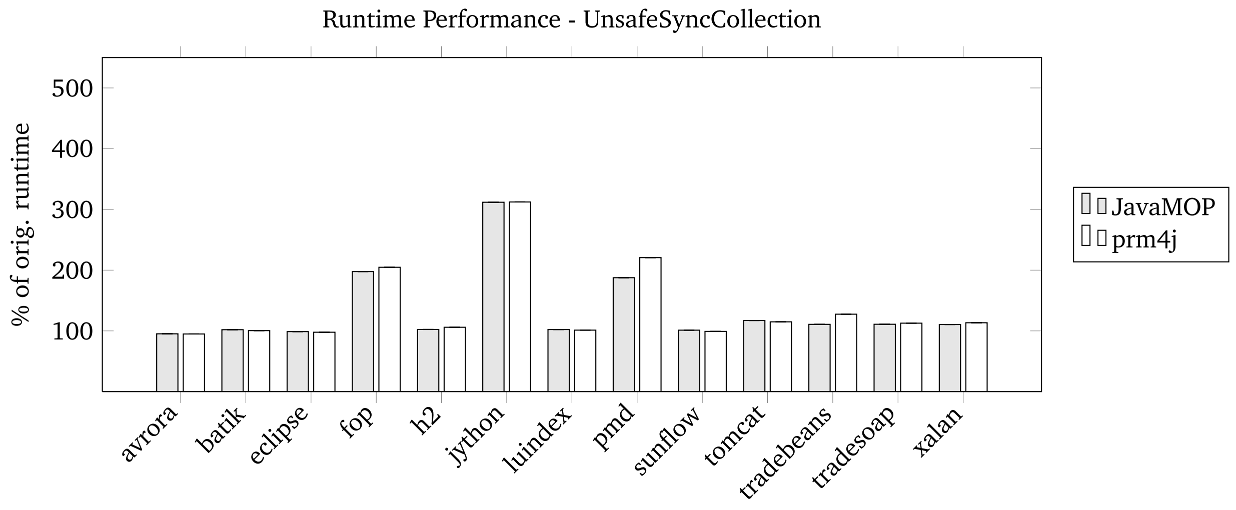


Figure 5.2.: Comparison of runtime performance in percent of original runtime with confidence level of 99% (pt.1)



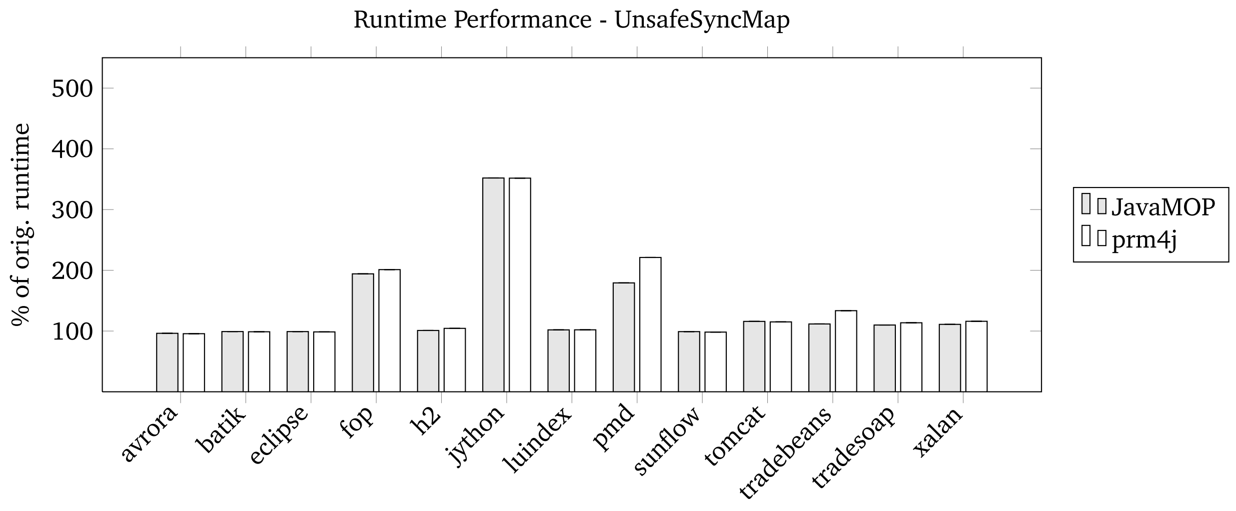


Figure 5.3.: Comparison of runtime performance in percent of original runtime with confidence level of 99% (pt.2)

	HasNext		UnsafeIterator		UnsafeMapIterator		UnsafeSyncCollection		UnsafeSyncMap	
Benchmark	MOP	prm4j	MOP	prm4j	MOP	prm4j	MOP	prm4j	MOP	prm4j
avrora	133.5	159.5	218.7	270.8	194.1	120.4	95.2	94.9	96.3	95.6
batik	103.1	103.6	112	112.1	110.1	102.7	102	100.4	99.2	98.8
eclipse	99.2	97.8	99.4	98.4	99.3	98.2	98.8	97.8	99.1	98.6
fop	236.8	257.8	324.9	280.3	266.3	178.5	197.7	204.7	194.2	201.1
h2	120.9	124.9	128.4	124.4	130.7	115	102.3	106	100.9	104.4
jython	269.6	268.6	376.1	372.6	532.6	515.1	311.8	312.3	352	351.7
luindex	102.2	101.9	102.8	102.8	102.3	102	102.2	101.2	101.9	102
pmd	202.3	245.1	347.2	436.1	315.4	206.5	187.6	220.6	179.3	221.1
sunflow	102.8	100	100.7	100.4	102.9	102.2	101.2	99.1	99	98.3
tomcat	115	113	117.4	118.2	118.4	117.7	117.1	114.9	115.9	115
tradebeans	309.3	360.7	359.9	372.6	413.1	284.5	110.7	127.5	111.6	133.5
tradesoap	152	157.1	243.6	238.2	206.7	159.3	110.9	112.7	109.9	113.6
xalan	109.7	113.2	135.7	141.5	106.3	125.1	110.5	113.4	111	116

Table 5.4.: Comparison of runtime performance in percent of original runtime

To summarize the results: prm4j achieves a runtime performance very similar to JavaMOP. On most benchmarks the runtime overhead is almost the same. On some benchmarks the benchmark is sometimes worse, sometimes better. It can be observed that the overhead is comparably greater when monitoring properties with small numbers of parameters like HasNext. The best results could be achieved when monitoring the most complex property regarding the number of parameters and monitor states, namely, UnsafeMapIterator.

Slightly shorter runtimes as compared to the baseline can be observed throughout the eclipse benchmark as well as with avrora and sunflow in conjunction with UnsafeSync{Map, Collection}. This has to be explained with positive effects of the synchronized monitoring on the garbage collection. Similar observations have been reported by other researchers [20].

5.2.3 Memory Consumption

The memory consumption is assessed by determining the mean and peak memory usage during monitoring. In general, we expect JavaMOP to have an advantage on memory consumption because it can encode information in generated code which we have to encode using conventional data structures. Memory usage is measured during each iteration by retrieving the currently used memory. The used memory is calculated by calling `Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()` every 100 events. It could be observed that the overhead of the load-time weaving significantly increases the memory consumption in the first iteration. The applied methodology described in Section 5.1.2 ensures that the first iteration is omitted. Because memory consumption is not sensible in reaching a steady-state we have increased the window of memory measurements dynamically to include all iterations except the first one. The reason is that some benchmarks show less memory consumption in the second iteration as compared to the later iterations. This affects prm4j and, to a slightly stronger degree, JavaMOP. Since all strong references to the used data structures are removed, the increase must be due to the peculiarities of the JVM memory management. A correlating increase of the measured runtimes could not be observed.

The memory peaks and mean memory consumption are depicted in Table 5.5 and 5.6.

The max and mean values were first averaged per invocation followed by the calculation of the confidence interval for all invocations. The values represent the mean of the confidence interval. The confidence interval width can be found in Appendix C. Due to the sufficient number of invocations the intervals do not overlap. Therefore, the differences between the means are statistically significant with a probability of 0.01 they are results of random errors.

		HasNext		UnsafeIterator		UnsafeMapIterator		UnsafeSyncCollection		UnsafeSyncMap	
Benchmark	Baseline	MOP	prm4j	MOP	prm4j	MOP	prm4j	MOP	prm4j	MOP	prm4j
avrora	29	64.8	69.2	113.8	95.4	124	68.5	8.8	8.9	8.8	8.8
batik	54.3	71.2	73.2	72	69.3	72.9	59.2	59.2	57.2	52	52.7
eclipse	218.4	89.2	86.9	90.1	89.8	88.9	86.8	29.3	29	29.4	29
fop	35.2	83.8	92.2	117.9	100.2	102.2	69.4	74	68.1	73.6	67.2
h2	272.5	316.5	320.2	346.8	333.3	380.2	282.9	221.1	220.5	221.2	220.3
jython	47.5	76.5	77	99.4	85.1	86.2	83	77.4	77.5	50.1	50
luindex	26.2	18.9	18.8	24.7	24.7	19.1	19	11.3	11.6	11.2	11.2
pmd	55.9	109.6	118.8	203.2	285.8	269.2	88.8	95	70.8	91.8	72.9
sunflow	32.4	36.7	37.7	36.9	37.2	37.1	37.1	7.7	7.8	7.8	7.7
tomcat	34	61.7	60.2	65.4	63.9	66.5	62.1	39.2	39.3	39.4	39.4
tradebeans	80.8	380.4	433.6	468.9	462.7	522	236.6	161.6	162.3	163.3	163.5
tradesoap	247.1	332.2	412.5	529.3	552.7	484.9	322.1	157.6	158.4	159.3	160.5
xalan	28.8	15.1	15	32.6	32.6	32.7	32.7	15.2	15.1	15.3	15.3

Table 5.5.: Peak memory consumption in megabyte sampled every 100 events

Considering HasNext, the values for memory consumption correlate in nearly all cases with the values for runtime overhead. As observed in runtime performance results for HasNext, JavaMOP outperforms prm4j in most of the cases. prm4j uses less memory in only four benchmarks. When monitoring with UnsafeIterator, the memory footprint of prm4j is smaller in all cases except pmd when compared to JavaMOP. An interesting result is that this is true for avrora, although the runtime overhead is larger than JavaMOP. This contradicts any general thesis of correlation of runtime overhead and memory overhead. The most probable explanation is the eager memory management in prm4j. This results in a higher runtime overhead for the collection of obsolete objects with positive impacts on memory consumption. With UnsafeMapIterator, prm4j can outperform JavaMOP on all benchmarks with a mean memory usage under 50% of JavaMOP's mean memory usage when monitoring pmd and tradebeans. Peak memory usage has similar proportions.

A reason for this difference in performance can be derived from additional statistics which were recorded during measurements of memory usage. The Table 5.7 depicts the created and collected bindings and monitors during monitoring with UnsafeMapIterator. *Orphaned monitors*, as described in Section 4.3.7, are monitors where the associated node has been garbage collected. Note that currently we can only detect monitors as orphaned or collected, which were removed in the according step of memory management. Hence, these monitors are removed some time after their associated node has been garbage collected. If a monitor is only strongly referenced by the NodeRef of its associated node, it is collected together with the node without any additional management step and is, therefore, not counted as orphaned monitor.

Benchmark	Baseline	HasNext MOP	HasNext prm4j	UnsafeIterator MOP	UnsafeIterator prm4j	UnsafeMapIterator MOP	UnsafeMapIterator prm4j	UnsafeSyncCollection MOP	UnsafeSyncCollection prm4j	UnsafeSyncMap MOP	UnsafeSyncMap prm4j
avrora	15.3	42.9	49.9	81.9	68.6	88.5	42.3	8.8	8.9	8.8	8.8
batik	39.7	42.7	41.1	42.7	43.9	45.7	39.8	42.8	42.5	52	52.7
eclipse	85	58	56.9	58.2	57.5	58	56.8	29.3	29	29.4	29
fop	20.9	55.6	61.4	78.4	69.6	69.9	48.7	49	46	48.7	45.5
h2	228.9	256.9	262.9	277.2	268.6	284	243.6	221.1	220.5	221.2	220.3
jython	32.6	67.9	68.2	80.2	74.1	76.7	73.6	69.3	68.7	50.1	50
luindex	13.5	13.3	13.3	16.9	17.1	13.6	13.5	11.3	11.6	11.2	11.2
pmd	34.9	73.4	70.8	138.2	177.9	169.5	62.9	66.5	54.7	64	54.7
sunflow	18.1	23.5	24.1	23.7	23.9	23.8	23.7	7.7	7.8	7.8	7.7
tomcat	34	50.4	49.5	52.4	51.9	56.2	52.2	39.2	39.3	39.4	39.4
tradebeans	20.3	286.9	312.4	354.9	338.9	401.1	212.3	161.6	162.3	163.3	163.5
tradesoap	128.4	237.6	261.7	381.3	377.7	340.1	223.1	157.6	158.4	159.3	160.5
xalan	15.1	15.1	15	24.1	24	24.2	24.2	15.2	15.1	15.3	15.3

Table 5.6.: Mean memory consumption in megabyte sampled every 100 events

In most benchmarks, the numbers of created monitors are considerably low although we register high numbers of bindings. This pattern suggests, that most of the events are either no creation events or belong to the same trace slice. Recall that the UnsafeMapIterator has event definitions for updates on maps and iterators which are registered as bindings. As these operations should be quite common in the target applications high numbers of events and bindings are to be expected. prm4j does not create monitors for this bindings because they are not carried by creation events.

We have not counted the number of created monitors in JavaMOP in this evaluation. Jin and Meredith [20] report statistics for RV which is the successor of JavaMOP (see also Section 2.1.5) showing larger numbers of created monitors for UnsafeMapIterator. Therefore, it is probable that JavaMOP has similar numbers and creates more monitors then prm4j, leading to higher overhead. As described in Section 5.1.5, it is confirmed by the designers of JavaMOP that it implements a similar optimization regarding creation events as prm4j. Hence, without deeper analysis we can not provide a reason for the deviating performance of JavaMOP.

Benchmark	Events	Runtime (%)	Memory (MB) Mean	Memory (MB) Max	Bindings Created	Bindings Collected	Monitors Created	Monitors Orphaned	Monitors Collected
avrora	1,260,000	120	42.3	68.5	1,010,000	945,000	15	10	5
batik	53,700	103	39.8	59.2	32,900	10,900	119	8.72	4.2
eclipse	7,290	98.2	56.8	86.8	1,940	955	67	0	0
fop	491,000	178	48.7	69.4	170,000	128,000	67	30	2.11
h2	1,750,000	115	244	283	402,000	342,000	4	0.241	0
jython	145,000	515	73.6	83	27,500	20,900	45	0	0
luindex	339	102	13.5	19	145	2.64	8	0	0
pmd	4,310,000	206	62.9	88.8	909,000	867,000	19,800	14,900	5,990
sunflow	1,280,000	102	23.7	37.1	82,800	82,500	1	0	0
tomcat	142,000	118	52.2	62.1	16,100	15,100	213	40.9	20
tradebeans	14,800,000	285	212	237	4,320,000	4,260,000	15.1	3.9	1.95
tradesoap	8,200,000	159	223	322	2,610,000	2,510,000	78,100	50,800	25,400
xalan	126,000	125	24.2	32.7	11,900	11,500	1	0.0933	0

Table 5.7.: Statistics for prm4j and UnsafeMapIterator

Memory consumption for UnsafeSyncCollection is nearly identical as compared to UnsafeSyncMap. prm4j and JavaMOP produce comparable results for mean memory usage and peak memory usage for nearly all benchmarks. When monitoring pmd, the memory footprint of prm4j is smaller as compared to JavaMOP while the runtime overhead is larger. We suspect this to be because of eager memory management in prm4j.

Statistics for all properties, including UnsafeSyncCollection and UnsafeSyncMap, can be found in Appendix D. It can be derived from these statistics, that after monitoring with UnsafeSyncCollection or UnsafeSyncMap counts for bindings and monitors are zero in many cases. This is because either no events were passed to the parametric monitor or no creation event was registered. prm4j implements a simple rule which activates the parametric monitor only after the first creation event was encountered. This has the effect in some cases that bindings are not stored even when events are registered. This technique was first described by Jin [19] and implemented in JavaMOP as well.

6 Future Work

In this chapter we summarize possible future work on prm4j or its underlying theory.

6.1 Formalism-Dependent Runtime Optimizations

Following the approach of Chen and Roşu [11], the core algorithm used in our approach is independent of a specific formalism. In our implementation we focused on the specification of properties using finite-state machines. Consequently, the monitors associated to the trace slices are implemented internally as finite-state machines. A finite-state machine has chances to reach the accepting state if it is expected that future events on the path to the accepting state may be encountered. Hence, aliveness of a monitor ideally depends on the state of the FSM and on possible future events.

Formalism-independence has the consequence that information about the FSM monitor state, although internally available, can not be exploited in the algorithm. During runtime, the state of a monitor is not accessible, since conceptually it has to be considered a black box. Monitor aliveness is, therefore, calculated depending on the parameter set which is bound by the monitor and on the expiration of the used bindings. Through analysis of the FSM pattern it is known which parameter sets correlate with which states. Often parameter sets can correlate with multiple states, therefore, accurate knowledge about the state can not be derived at runtime. This gets increasingly problematic with a rising number of monitor states in the specification. Consequently, predictions about monitor aliveness may be to optimistic in certain cases, causing runtime overhead.

Hence, we consider the exploration of formalism dependent optimizations as potentially fruitful. The general structure of object-based indexing can be preserved while augmenting the API with an useful abstraction to exploit information specific to certain monitor types.

6.2 Memory Management of Bindings Used in Match Handlers

Weakly binding parameters do not provide the guarantee that the bound value is still accessible when a match is reported. Therefore, match handlers can not expect the bound objects to be stored in memory when the handler code is executed. To provide this guarantee, values have to be bound using strong references. As objects from the monitored application are held in memory from the moment they are first passed in a parametric event, this leads to excessive memory consumption and a high runtime overhead.

A reference management depending on monitor aliveness would be more efficient. If the binding is used in a match handler of a monitor which has the chance of reaching an accepting state, the bound object has to be referenced strongly. If no such monitor exists, the bound object is just referenced weakly. A management of references to parameter values depending on monitor aliveness is not implemented neither in prm4j nor JavaMOP. Tracematches [2], on the other hand, introduces *collectableWeakRefs* which allow bindings to be marked as eligible for garbage collection when a monitor is detected to be unable to reach a matching state. Hence, guarantees for the case of a trace match can be efficiently provided. Following the argumentation of Avgustinov et al. [2], a fine-granular management of references conflicts with a strong interpretation of formalism independence as the state of the finite-state machine can not be exploited. But even if this information can be used, a challenge will be to develop a scheme how to propagate this information through the indexing tree efficiently. At the time of writing, we regard this scheme to be possible since references to bindings are maintained in each monitor. The binding, still a weak reference, just needs to keep track of the *number* of dependent alive monitors. If this number is greater then zero, the bound object is additionally strongly referenced, prolonging its lifetime. If a monitor terminates, it just needs to propagate this information to the binding, before being garbage collected, decrementing the usage counter. Currently, our memory management model is to eager in pruning branches together with monitors, so monitors may be garbage collected before propagating information about their expiration to all their bindings. An improvement is, therefore, left to future work.

6.3 Efficient Runtime Monitoring with Multiple Properties

Our library was developed with the possibility of parallel monitoring of multiple properties in mind. Many useful insights could be drawn from work of Jin and Meredith [21, 19], regarding optimizations allowing efficient monitoring of multiple properties in JavaMOP. Thanks to this work, we could hopefully avoid design decisions seriously impairing the development of a future scalable version of prm4j. E.g., bindings are agnostic of the parameters they are bound to allowing to reuse them for monitoring other properties. As Jin and Meredith observed, maintaining bindings is a major source of overhead in JavaMOP. This bindings can be used in a centralized binding store. Future work, therefore, can be funded on related work in JavaMOP, e.g., a single binding store, combined indexing trees for multiple properties etc.

6.4 Evaluation of the Performance Impact of Blocking

As sketched in the motivation, the work on this thesis started with the development of a lock-free indexing scheme for monitoring. We learned from this work that the object-based indexing algorithm underlying prm4j and JavaMOP is very hard to parallelize. We could not find effective means for thread-safe non-blocking mutations of the indexing data structure in MOPBox without having impact on the semantics of match handling. E.g., buffering of timestamped events was tried with the logical consequence of delayed execution of recovery code in the case of a match. The development of a micro-locking scheme as found in the ConcurrentHashMap implemented in the Java JDK was rejected for the indexing algorithms in MOPBox. As the development on prm4j focused on providing a fast blocking solution, safe mutations of the indexing tree by multiple threads were not considered. Therefore, the parametric monitor used in prm4j locks the entire indexing tree globally. This is the solution which is also found in JavaMOP. Therefore, it may be the case that a non-blocking indexing scheme for this type of algorithm is in fact infeasible. However, it is expected that this global lock will have effects on runtime performance when monitoring highly parallel applications. Hence, future work considering synchronization may be fruitful if performance bottlenecks are properly analyzed and parallelize computations can be found. When monitoring multiple properties, a single non-blocking binding store could be more efficient then a blocking solution. It would also be useful to explore possibilities of implementing the memory management as a background task managed by dedicated threads.

7 Conclusion

We have presented the implementation and underlying theory of prm4j[1], a Java library for the efficient runtime monitoring of parametric properties. It allows to specify and monitor parametric properties over traces of events which bind Java objects. These properties can be specified directly in Java code without the dependency on a specific formalism. A specification API using finite-state machines is provided. The parametric monitor for a property can be instantiated without code generation directly in Java. Parametric events are plain Java objects which can be generated from target applications instrumented with AspectJ or other sources.

We have evaluated the runtime performance and memory footprint of prm4j using the established DaCapo benchmark suite. This suite comprises realistic, general purpose Java applications that are widely used. The monitoring overhead was compared to JavaMOP [13], one of the most optimized parametric monitoring frameworks. JavaMOP relies on the generation of highly optimized code. Our results show that the runtime performance and memory consumption of prm4j and JavaMOP are comparable. Very simple properties can be monitored more efficiently by JavaMOP. When monitoring more complex properties, the overhead generated by prm4j is comparably lower.

These results are very promising. A library-based solution allows the comfortable and dynamic specification of parametric monitors with seamless integration into existing tool chains. Our library delivers this comfort and integration with a very low runtime overhead. Users do not need to hesitate to use the library for monitoring real applications.

Our implementation is based on the parametric trace slicing approach proposed by Chen et al. [14]. This approach allows to stay independent of specific property-specification formalisms by dispatching parametric events to monitors considered as black-boxes. Hence, the library can be easily extended by providing additional formalisms. Future extensions and optimizations are simplified by the extensive test suite provided with prm4j and the evaluation framework prm4j-eval. prm4j-eval[2] implements a statistically rigorous performance evaluation methodology, providing a valuable tool for any optimization of prm4j in the future.

[1] Source code of prm4j : `https://github.com/parzonka/prm4j`

[2] Source code of prm4j-eval : `https://github.com/parzonka/prm4j-eval`

Appendices

A prm4j Property Specifications

```
public class FSM_HasNext {

    public final Alphabet alphabet = new Alphabet();

    public final Parameter<Iterator> i = alphabet.createParameter("i", Iterator.class);

    public final Symbol1<Iterator> hasNext = alphabet.createSymbol1("hasNext", i);
    public final Symbol1<Iterator> next = alphabet.createSymbol1("next", i);

    public final FSM fsm = new FSM(alphabet);

    public final  MatchHandler matchHandler = MatchHandler.NO_OP;

    public final FSMState initial = fsm.createInitialState();
    public final FSMState safe = fsm.createState();
    public final FSMState error = fsm.createAcceptingState(matchHandler);

    public FSM_HasNext() {
        initial.addTransition(hasNext, safe);
        initial.addTransition(next, error);
        safe.addTransition(hasNext, safe);
        safe.addTransition(next, initial);
        error.addTransition(next, error);
        error.addTransition(hasNext, safe);
    }
}

////////////////////////////////////////////////////////////////////////

public aspect HasNext extends Prm4jAspect {

    final FSM_HasNext fsm;
    final ParametricMonitor pm;

    public HasNext() {
    fsm = new FSM_HasNext();
    pm = ParametricMonitorFactory.createParametricMonitor(new FSMSpec(fsm.fsm));
    }

    after(Iterator i) : call(* Iterator.hasNext()) && target(i) && prm4jPointcut() {
    pm.processEvent(fsm.hasNext.createEvent(i));
    }

    before(Iterator i) : call(* Iterator.next()) && target(i)  && prm4jPointcut() {
    pm.processEvent(fsm.next.createEvent(i));
    }

}
```

Listing A.1: HasNext property for prm4j

```
public aspect UnsafeIterator extends Prm4jAspect {

    private final ParametricMonitor pm;
    private final Alphabet alphabet = new Alphabet();

    // parameters
    private final Parameter<Collection> c = alphabet.createParameter("c", Collection.class);
    private final Parameter<Iterator> i = alphabet.createParameter("i", Iterator.class);

    // symbols
    private final Symbol2<Collection, Iterator> createIter =
        alphabet.createSymbol2("createIter", c, i);
    private final Symbol1<Collection> updateColl = alphabet.createSymbol1("updateColl", c);
    private final Symbol1<Iterator> useIter = alphabet.createSymbol1("useIter", i);

    // match handler
    public final  MatchHandler matchHandler = MatchHandler.SYS_OUT;

    final FSM fsm = new FSM(alphabet);

    public UnsafeIterator() {

        // fsm states
        final FSMState initial = fsm.createInitialState();
        final FSMState s1 = fsm.createState();
        final FSMState s2 = fsm.createState();
        final FSMState error = fsm.createAcceptingState(matchHandler);

        // fsm transitions
        initial.addTransition(updateColl, initial);
        initial.addTransition(createIter, s1);
        s1.addTransition(useIter, s1);
        s1.addTransition(updateColl, s2);
        s2.addTransition(updateColl, s2);
        s2.addTransition(useIter, error);

        // parametric monitor creation
        pm = ParametricMonitorFactory.createParametricMonitor(new FSMSpec(fsm));

    }

    after(Collection c) returning (Iterator i) : (call(Iterator Collection+.iterator())
        && target(c)) && prm4jPointcut() {
    pm.processEvent(createIter.createEvent(c, i));
    }

    after(Collection c) : ((call(* Collection+.remove*(..)) || call(* Collection+.add*(..)))
        && target(c)) && prm4jPointcut() {
    pm.processEvent(updateColl.createEvent(c));
    }

    before(Iterator i) : (call(* Iterator.next()) && target(i)) && prm4jPointcut() {
    pm.processEvent(useIter.createEvent(i));
    }

}
```

Listing A.2: UnsafeIterator property for prm4j

```
public class FSM_UnsafeMapIterator {

    public final Alphabet alphabet = new Alphabet();

    public final Parameter<Map> m = alphabet.createParameter("m", Map.class);
    public final Parameter<Collection> c = alphabet.createParameter("c", Collection.class);
    public final Parameter<Iterator> i = alphabet.createParameter("i", Iterator.class);

    public final Symbol2<Map, Collection> createColl = alphabet.createSymbol2("createColl", m, c);
    public final Symbol2<Collection, Iterator> createIter =
        alphabet.createSymbol2("createIter", c, i);
    public final Symbol1<Map> updateMap = alphabet.createSymbol1("updateMap", m);
    public final Symbol1<Iterator> useIter = alphabet.createSymbol1("useIter", i);

    public final FSM fsm = new FSM(alphabet);
    public final MatchHandler matchHandler = MatchHandler.NO_OP;

    public final FSMState initial = fsm.createInitialState();
    public final FSMState s1 = fsm.createState();
    public final FSMState s2 = fsm.createState();
    public final FSMState s3 = fsm.createState();
    public final FSMState error = fsm.createAcceptingState(matchHandler);

    public FSM_UnsafeMapIterator() {
        initial.addTransition(createColl, s1); // creation event
        initial.addTransition(updateMap, initial); // self-loop
        initial.addTransition(useIter, initial); // self-loop
        initial.addTransition(createIter, initial); // self-loop
        s1.addTransition(updateMap, s1);
        s1.addTransition(createIter, s2);
        s2.addTransition(useIter, s2);
        s2.addTransition(updateMap, s3);
        s3.addTransition(updateMap, s3);
        s3.addTransition(useIter, error);
    }
}

////////////////////////////////////////////////////////////////////////

public aspect UnsafeMapIterator extends Prm4jAspect {

    final FSM_UnsafeMapIterator fsm;
    final ParametricMonitor pm;

    public UnsafeMapIterator() {
        fsm = new FSM_UnsafeMapIterator();
        pm = ParametricMonitorFactory.createParametricMonitor(new FSMSpec(fsm.fsm));
    }

    after(Map map) returning (Collection c) : (call(* Map.values()) || call(* Map.keySet()))
              && target(map) && prm4jPointcut() {
        pm.processEvent(fsm.createColl.createEvent(map, c));
    }
    after(Collection c) returning (Iterator i) : call(* Collection.iterator())
              && target(c) && prm4jPointcut() {
        pm.processEvent(fsm.createIter.createEvent(c, i));
    }
    before(Iterator i) : call(* Iterator.next()) && target(i) && prm4jPointcut() {
        pm.processEvent(fsm.useIter.createEvent(i));
    }
    after(Map map) : (call(* Map.put*(..)) || call(* Map.putAll*(..)) || call(* Map.clear())
            || call(* Map.remove*(..))) && target(map) && prm4jPointcut(){
        pm.processEvent(fsm.updateMap.createEvent(map));
    }
}
```

Listing A.3: UnsafeMapIterator property for prm4j

```java
public class FSM_UnsafeSyncCollection {

    public final Alphabet alphabet = new Alphabet();

    public final Parameter<Collection> c = alphabet.createParameter("c", Collection.class);
    public final Parameter<Iterator> i = alphabet.createParameter("i", Iterator.class);

    public final Symbol1<Collection> sync = alphabet.createSymbol1("sync", c);
    public final Symbol2<Collection, Iterator> asyncCreateIter =
        alphabet.createSymbol2("asyncCreateIter", c, i);
    public final Symbol2<Collection, Iterator> syncCreateIter =
        alphabet.createSymbol2("syncCreateIter", c, i);
    public final Symbol1<Iterator> accessIter = alphabet.createSymbol1("accessIter", i);

    public final FSM fsm = new FSM(alphabet);

    public final MatchHandler matchHandler = MatchHandler.NO_OP;

    public final FSMState initial = fsm.createInitialState();
    public final FSMState s1 = fsm.createState();
    public final FSMState s2 = fsm.createState();
    public final FSMState error = fsm.createAcceptingState(matchHandler);

    public FSM_UnsafeSyncCollection() {
        initial.addTransition(sync, s1); // creation event
        initial.addTransition(asyncCreateIter, initial); // self-loop
        initial.addTransition(syncCreateIter, initial); // self-loop
        initial.addTransition(accessIter, initial); // self-loop
        s1.addTransition(asyncCreateIter, error);
        s1.addTransition(syncCreateIter, s2);
        s2.addTransition(accessIter, error);
    }
}

///////////////////////////////////////////////////////////////////////////

public aspect UnsafeSyncCollection extends Prm4jAspect {

    final FSM_UnsafeSyncCollection fsm;
    final ParametricMonitor pm;

    public UnsafeSyncCollection() {
        fsm = new FSM_UnsafeSyncCollection();
        pm = ParametricMonitorFactory.createParametricMonitor(new FSMSpec(fsm.fsm));
    }


    after() returning (Collection c) : (call(* Collections.synchr*(..))) && prm4jPointcut() {
        pm.processEvent(fsm.sync.createEvent(c));
    }
    after(Collection c) returning (Iterator i) : (call(* Collection+.iterator()) && target(c)
            && if(Thread.holdsLock(c))) && prm4jPointcut() {
        pm.processEvent(fsm.syncCreateIter.createEvent(c, i));
    }
    after(Collection c) returning (Iterator i) : (call(* Collection+.iterator()) && target(c)
            && if(!Thread.holdsLock(c))) && prm4jPointcut() {
        pm.processEvent(fsm.asyncCreateIter.createEvent(c, i));
    }
    final Condition threadHoldsNoLockOnCollection = new Condition() {
        @Override
        public boolean eval() {
            return !Thread.holdsLock(getParameterValue(fsm.c));
        }
    };
    before(Iterator i) : (call(* Iterator.*(..)) && target(i)) && prm4jPointcut() {
        pm.processEvent(fsm.accessIter.createConditionalEvent(i, threadHoldsNoLockOnCollection));
    }
}
```

Listing A.4: UnsafeSyncCollection property for prm4j

```java
public class FSM_UnsafeSyncMap {

    public final Alphabet alphabet = new Alphabet();

    public final Parameter<Map> m = alphabet.createParameter("m", Map.class);
    public final Parameter<Collection> c = alphabet.createParameter("c", Collection.class);
    public final Parameter<Iterator> i = alphabet.createParameter("i", Iterator.class);

    public final Symbol1<Map> sync = alphabet.createSymbol1("sync", m);
    public final Symbol2<Map, Collection> createSet = alphabet.createSymbol2("createSet", m, c);
    public final Symbol2<Collection, Iterator> asyncCreateIter = alphabet.createSymbol2("
        asyncCreateIter", c, i);
    public final Symbol2<Collection, Iterator> syncCreateIter = alphabet.createSymbol2("
        syncCreateIter", c, i);
    public final Symbol1<Iterator> accessIter = alphabet.createSymbol1("accessIter", i);

    public final FSM fsm = new FSM(alphabet);

    public final MatchHandler matchHandler = MatchHandler.NO_OP;

    public final FSMState initial = fsm.createInitialState();
    public final FSMState s1 = fsm.createState();
    public final FSMState s2 = fsm.createState();
    public final FSMState s3 = fsm.createState();
    public final FSMState error = fsm.createAcceptingState(matchHandler);

    public FSM_UnsafeSyncMap() {
        initial.addTransition(sync, s1); // creation event
        initial.addTransition(createSet, initial); // self-loop
        initial.addTransition(asyncCreateIter, initial); // self-loop
        initial.addTransition(syncCreateIter, initial); // self-loop
        initial.addTransition(accessIter, initial); // self-loop
        s1.addTransition(createSet, s2);
        s2.addTransition(asyncCreateIter, error);
        s2.addTransition(syncCreateIter, s3);
        s3.addTransition(accessIter, error);
    }
}

////////////////////////////////////////////////////////////////////////

public aspect UnsafeSyncMap extends Prm4jAspect {

    final FSM_UnsafeSyncMap fsm;
    final ParametricMonitor pm;

    public UnsafeSyncMap() {
        fsm = new FSM_UnsafeSyncMap();
        pm = ParametricMonitorFactory.createParametricMonitor(new FSMSpec(fsm.fsm));
    }

    final Condition threadHoldsLockOnCollection = new Condition() {
        @Override
        public boolean eval() {
            return Thread.holdsLock(getParameterValue(fsm.c));
        }
    };

    final Condition threadHoldsNoLockOnCollection = new Condition() {
        @Override
        public boolean eval() {
            return !Thread.holdsLock(getParameterValue(fsm.c));
        }
    };

    after () returning (Map syncMap) : (call(* Collections.synchr*(..))) && prm4jPointcut() {
    pm.processEvent(fsm.sync.createEvent(syncMap));
    }

    after (Map syncMap) returning (Set mapSet) : (call(* Map+.keySet()) && target(syncMap)) &&
        prm4jPointcut() {
        pm.processEvent(fsm.createSet.createEvent(syncMap, mapSet));
    }

    after(Set mapSet) returning (Iterator iter) : (call(* Collection+.iterator()) && target(mapSet))
        && prm4jPointcut() {
        pm.processEvent(fsm.syncCreateIter.createConditionalEvent(mapSet, iter,
            threadHoldsLockOnCollection));
        pm.processEvent(fsm.asyncCreateIter.createConditionalEvent(mapSet, iter,
            threadHoldsNoLockOnCollection));
    }

    before(Iterator i) : (call(* Iterator.*(..)) && target(i)) && prm4jPointcut() {
        pm.processEvent(fsm.accessIter.createConditionalEvent(i, threadHoldsNoLockOnCollection));
    }
}
```

Listing A.5: UnsafeSyncMap property for prm4j

B JavaMOP Property Specifications

These specifications were used for evaluation. They are included in JavaMOP 3.0.0. Note that the call to `System.out.println` is removed later in the modified evaluation aspect.

```
package mop;

import java.io.*;
import java.util.*;

// This property specifies that a program does
// not call the hasnext method  before the next
// method of an iterator.
// This property is borrowed from tracematches
// (see ECOOP'07 http://abc.comlab.ox.ac.uk/papers)

full-binding HasNext(Iterator i) {
   event hasnext after(Iterator i) :
      call(* Iterator.hasNext()) && target(i) {}
   event next before(Iterator i) :
      call(* Iterator.next()) && target(i) {}

   fsm :
     start [
        next -> unsafe
        hasnext -> safe
     ]
     safe [
        next -> start
        hasnext -> safe
     ]
     unsafe [
        next -> unsafe
        hasnext -> safe
     ]

     alias match = unsafe

   @match {
      System.out.println("next called without hasNext!");
   }
}
```

Listing B.1: HasNext.mop

```
package mop;

import java.io.*;
import java.util.*;

// The UnsafeIterator property is designed
// to match a case where a Collection that
// is in the process of iteration is modified
// and iteration continues.

UnsafeIterator(Collection c, Iterator i) {

        event create after(Collection c)
              returning(Iterator i) :
              call(Iterator Collection+.iterator()) && target(c) {}
        event updatesource after(Collection c) :
              (call(* Collection+.remove*(..))
              || call(* Collection+.add*(..)) ) && target(c) {}
        event next before(Iterator i) :
              call(* Iterator.next()) && target(i) {}

        ere : create next* updatesource updatesource* next

        @match {
           System.out.println("improper iterator usage");
        }
}
```

Listing B.2: UnsafeIterator.mop

```
package mop;

import java.io.*;
import java.util.*;


// UnsafeMapIterator is similar to
// UnsafeIterator.  The biggest difference
// is that a key collection of the map
// is created, and the iterator is created
// from collection.  This offers a larger
// challenge in JavaMOP, because the monitor
// creation events do not contain all the
// parameters (because the collection will be
// created before the iterator, and, in fact,
// many iterators can be created from one map).

full-binding UnsafeMapIterator(Map map, Collection c, Iterator i){
  event createColl after(Map map)
              returning(Collection c) :
              (call(* Map.values())
              || call(* Map.keySet()))
              && target(map) {}
  event createIter after(Collection c)
              returning(Iterator i) :
        call(* Collection.iterator())
              && target(c) {}
  event useIter before(Iterator i) :
        call(* Iterator.next())
              && target(i) {}
  event updateMap after(Map map) :
        (call(* Map.put*(..))
              || call(* Map.putAll*(..))
              || call(* Map.clear())
              || call(* Map.remove*(..)))
              && target(map) {}

    ere : createColl updateMap* createIter useIter* updateMap updateMap* useIter

  @match{
           System.out.println("unsafe iterator usage!");
  }
}
```

Listing B.3: UnsafeMapIterator.mop

```
package mop;

import java.io.*;
import java.util.*;

// The SafeSyncCollection property is designed
// to match a case where either a collection
// is synchronized and an non-synchronized
// iterator is created for the collection, or
// a synchronized iterator is created, but
// accessed in an unsynchronized manner.

SafeSyncCollection(Object c, Iterator iter) {
  Object c;

  creation event sync after() returning(Object c) :
    call(* Collections.synchr*(..)) {
      this.c = c;
    }
  event syncCreateIter after(Object c)
                returning(Iterator iter) :
    call(* Collection+.iterator())
                && target(c) && if(Thread.holdsLock(c)){}
  event asyncCreateIter after(Object c) returning(Iterator iter) :
    call(* Collection+.iterator())
                && target(c)
                && if(!Thread.holdsLock(c)){}
  event accessIter before(Iterator iter) :
    call(* Iterator.*(..))
                && target(iter)
                && condition(!Thread.holdsLock(this.c)) {}

  ere : (sync asyncCreateIter)
            | (sync syncCreateIter accessIter)

  @match{
    System.out.println("pattern matched!");
  }
}
```

Listing B.4: UnsafeSyncCollection.mop

```
package mop;

import java.io.*;
import java.util.*;

// The SafeSyncMap property is designed
// to match a case where either a collection
// is synchronized and an non-synchronized
// iterator is created for the collection,
// or a synchronized iterator is created, but
// accessed in an unsynchronized manner. The
// difference from SafeSyncCollection is that
// a set must be created from the
// synchronized map.

SafeSyncMap(Map syncMap, Set+ mapSet, Iterator iter) {
  Map c;
  creation event sync after()
              returning(Map syncMap) :
    call(* Collections.synchr*(..)) {
      this.c = syncMap;
    }
  event createSet after(Map syncMap)
              returning(Set+ mapSet) :
          call(* Map+.keySet())
              && target(syncMap) {
              }
  event syncCreateIter after(Set+ mapSet)
              returning(Iterator iter) :
    call(* Collection+.iterator())
              && target(mapSet)
              && condition(Thread.holdsLock(c)){
              }
  event asyncCreateIter after(Set mapSet)
              returning(Iterator iter) :
    call(* Collection+.iterator())
              && target(mapSet)
              && condition(!Thread.holdsLock(c)) {
              }
  event accessIter before(Iterator iter) :
    call(* Iterator.*(..))
              && target(iter)
              && condition(!Thread.holdsLock(c)) {
              }

  ere : sync createSet
             (asyncCreateIter | (syncCreateIter accessIter))

  @match{
          System.out.println("synchronized collection accessed in non threadsafe manner!");
  }
}
```

Listing B.5: UnsafeSyncMap.mop

This property exemplifies the issue in the current JavaMOP implementation described in Section 5.2.1. It is a result of associating instance timestamps with sole bindings instead of associating them with instances.

```
package mop;

import java.io.*;
import java.util.*;

// This property exemplifies the issue in the current JavaMOP implementation.
// It is a result of associating instance timestamps with sole bindings instead of associating them with instan
//
// The following trace yields an incorrect result:
// e1<a1>
// e2<a2b1> : this slice a2b1 interferes with the slice for a1b1
// e2<a1b1> : no match is reported, which is incorrect

X(A a, B b) {

    event e1 before(A a) :
                call(* A.e1()) && target(a) {}
    event e2 after  (A a) returning (B b) :
          call(* A.e2()) && target(a) {}

        ere : e1 e2

        @match {
           System.out.println("e1 e2 detected!");
        }
}
```

Listing B.6: JavaMOP-Issue.mop

C Evaluation Confidence Interval Widths

	HasNext		UnsafeIterator		UnsafeMapIterator		UnsafeSyncCollection		UnsafeSyncMap	
Benchmark	MOP	prm4j	MOP	prm4j	MOP	prm4j	MOP	prm4j	MOP	prm4j
avrora	0.003	0.0006	0.0013	0.0013	0.0013	0.0008	0.0009	0.0007	0.0009	0.0007
batik	0.0003	0.0002	0.0003	0.0002	0.0002	0.0001	0.0001	0.0001	0.0002	0.0002
eclipse	0.0063	0.004	0.0059	0.0029	0.0041	0.0045	0.0073	0.0054	0.005	0.0058
fop	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0002	0.0001
h2	0.0008	0.001	0.0008	0.0008	0.0007	0.0007	0.0006	0.0012	0.0008	0.001
jython	0.0003	0.0004	0.0012	0.001	0.002	0.0004	0.0003	0.0004	0.0004	0.0003
luindex	0.0001	0	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
pmd	0.0008	0.0006	0.0004	0.0003	0.0005	0.0006	0.0005	0.0007	0.0006	0.0006
sunflow	0.0021	0.002	0.0028	0.0019	0.0014	0.0019	0.002	0.0027	0.0024	0.0028
tomcat	0.0026	0.002	0.0016	0.0025	0.0018	0.0028	0.002	0.0016	0.0016	0.0014
tradebeans	0.0004	0.001	0.0006	0.0008	0.001	0.0007	0.0005	0.0008	0.0006	0.0005
tradesoap	0.0009	0.0015	0.0044	0.0036	0.0021	0.0007	0.001	0.0022	0.0013	0.0015
xalan	0.002	0.0024	0.0019	0.0019	0.0014	0.0022	0.0022	0.0017	0.0019	0.0025

Table C.1.: Half confidence interval width in percent of original runtime with confidence level 99%

	HasNext		UnsafeIterator		UnsafeMapIterator		UnsafeSyncCollection		UnsafeSyncMap	
benchmark	MOP	prm4j	MOP	prm4j	MOP	prm4j	MOP	prm4j	MOP	prm4j
avrora	2.5304	0.9272	1.3441	1.2098	2.0467	0.9996	0.0153	0.0623	0.0153	0.0609
batik	1.0061	0.0539	0.4775	0.106	0.6419	0.0781	0.7314	0.0406	1.057	1.3781
eclipse	0.6362	1.3129	0.6429	1.2716	0.6813	1.3453	0.4807	1.3399	0.4492	1.2079
fop	0.254	0.3206	0.6257	1.4853	0.3631	0.6781	0.3275	0.605	0.2865	0.5468
h2	0.4274	0.2031	0.6424	0.2465	0.8328	0.1886	0.0283	0.0433	0.0387	0.0396
jython	0.0769	0.2298	0.1157	0.1749	0.0863	0.141	0.0516	0.1374	0.0335	0.078
luindex	0.0558	0.0719	0.0089	0.041	0.0466	0.0765	0.2171	0.4408	0.227	0.5717
pmd	0.9741	1.3278	3.1392	12.5529	2.9694	1.8062	0.653	1.3602	0.7227	1.2481
sunflow	0.0661	0.1186	0.0569	0.1126	0.0596	0.1278	0.0043	0.0017	0.0052	0.0047
tomcat	0.0747	0.287	0.0518	0.0809	0.0648	0.3382	0.0293	0.0401	0.0283	0.0268
tradebeans	1.8201	9.4859	4.8254	7.9507	5.1733	1.4542	0.142	0.1561	0.0575	0.1312
tradesoap	0.7159	7.7426	3.1997	9.041	5.1964	0.8807	0.6578	0.6101	0.6785	1.0592
xalan	0.011	0.0201	0.0068	0.017	0.0175	0.0126	0.0096	0.0195	0.0094	0.0093

Table C.2.: Half confidence interval width for mean memory consumption in megabyte with confidence level 99%

	HasNext		UnsafeIterator		UnsafeMapIterator		UnsafeSyncCollection		UnsafeSyncMap	
benchmark	MOP	prm4j	MOP	prm4j	MOP	prm4j	MOP	prm4j	MOP	prm4j
avrora	4.4153	1.5598	2.339	2.5283	3.1356	2.1991	0.0153	0.0623	0.0153	0.0609
batik	1.0344	0.2376	0.4899	0.2233	0.65	0.3202	0.731	0.1277	1.057	1.3781
eclipse	1.5489	3.5354	1.7003	2.9473	1.6303	3.171	0.4807	1.3399	0.4492	1.2079
fop	0.3397	0.7113	1.0194	1.9838	0.6646	0.7688	0.3678	0.642	0.314	0.5944
h2	1.751	2.1923	2.3679	3.9344	2.1038	4.4392	0.0283	0.0433	0.0387	0.0396
jython	0.0925	0.1988	0.4843	0.199	0.1123	0.138	0.0832	0.1202	0.0335	0.078
luindex	0.0515	0.0314	0.0142	0.052	0.0422	0.0382	0.2171	0.4408	0.227	0.5717
pmd	2.385	4.398	5.5297	17.5028	8.0299	2.1799	1.3467	1.2743	1.3698	1.3707
sunflow	0.0973	0.1422	0.0817	0.1584	0.0839	0.1915	0.0043	0.0017	0.0052	0.0047
tomcat	0.1087	0.2678	0.0971	0.1173	0.0949	0.3559	0.0293	0.0401	0.0283	0.0268
tradebeans	3.316	17.7956	8.0624	15.9132	8.1253	1.5461	0.2754	0.1591	0.0575	0.1325
tradesoap	1.8903	21.3662	4.9489	15.0501	10.4513	2.3571	0.6564	0.6101	0.6785	1.0592
xalan	0.0115	0.0209	0.0128	0.0193	0.0226	0.0245	0.0096	0.0195	0.0094	0.0093

Table C.3.: Half confidence interval width for peak memory consumption in megabyte with confidence level 99%

D Statistics for prm4j

Orphaned monitors are monitors where the associated node has been garbage collected.

			Memory (MB)		Bindings			Monitors	
Benchmark	Events	Runtime (%)	Mean	Max	Created	Collected	Created	Orphaned	Collected
avrora	1,510,000	159	49.9	69.2	909,000	826,000	909,000	0.0833	0.0833
batik	48,200	104	41.1	73.2	24,300	4,270	24,300	0	0
eclipse	10,900	97.8	56.9	86.9	940	636	940	0	0
fop	1,010,000	258	61.4	92.2	184,000	73,900	184,000	0	0
h2	2,690,000	125	263	320	384,000	320,000	384,000	0	0
jython	443,000	269	68.2	77	25,800	22,800	25,800	0.027	0.027
luindex	364	102	13.3	18.8	64	1.65	64	0	0
pmd	8,280,000	245	70.8	119	792,000	721,000	792,000	0.0222	0.0222
sunflow	2,660,000	100	24.1	37.7	101,000	100,000	101,000	0	0
tomcat	111,000	113	49.5	60.2	11,800	11,300	11,800	0.0625	0
tradebeans	21,800,000	361	312	434	3,840,000	3,640,000	3,840,000	0	0
tradesoap	8,590,000	157	262	413	1,830,000	1,820,000	1,830,000	0.0645	0
xalan	4	113	15	15	1	0	1	0	0

Table D.1.: Statistics for prm4j and HasNext

			Memory (MB)		Bindings			Monitors	
Benchmark	Events	Runtime (%)	Mean	Max	Created	Collected	Created	Orphaned	Collected
avrora	1,370,000	271	68.6	95.4	1,010,000	947,000	909,000	37,600	37,600
batik	125,000	112	43.9	69.3	43,200	11,200	24,300	3.04	2.37
eclipse	5,980	98.4	57.5	89.8	1,760	826	937	0	0
fop	705,000	280	69.6	100	250,000	130,000	7,740	713	712
h2	1,480,000	124	269	333	387,000	273,000	3,190	0	0
jython	715,000	373	74.1	85.1	115,000	95,500	3,060	10.3	10.3
luindex	4,320	103	17.1	24.7	1,400	229	64	0	0
pmd	6,450,000	436	178	286	1,360,000	1,190,000	553,000	304,000	294,000
sunflow	1,280,000	100	23.9	37.2	82,800	82,500	1	0	0
tomcat	99,500	118	51.9	63.9	20,700	19,200	11,800	3,450	3,450
tradebeans	11,100,000	373	339	463	3,970,000	3,820,000	132,000	48,200	48,200
tradesoap	10,500,000	238	378	553	3,760,000	3,700,000	850,000	496,000	496,000
xalan	1,310,000	141	24	32.6	15,900	15,700	1	0	0

Table D.2.: Statistics for prm4j and UnsafeIterator

			Memory (MB)		Bindings			Monitors	
Benchmark	Events	Runtime (%)	Mean	Max	Created	Collected	Created	Orphaned	Collected
avrora	1,260,000	120	42.3	68.5	1,010,000	945,000	15	10	5
batik	53,700	103	39.8	59.2	32,900	10,900	119	8.72	4.2
eclipse	7,290	98.2	56.8	86.8	1,940	955	67	0	0
fop	491,000	178	48.7	69.4	170,000	128,000	67	30	2.11
h2	1,750,000	115	244	283	402,000	342,000	4	0.241	0
jython	145,000	515	73.6	83	27,500	20,900	45	0	0
luindex	339	102	13.5	19	145	2.64	8	0	0
pmd	4,310,000	206	62.9	88.8	909,000	867,000	19,800	14,900	5,990
sunflow	1,280,000	102	23.7	37.1	82,800	82,500	1	0	0
tomcat	142,000	118	52.2	62.1	16,100	15,100	213	40.9	20
tradebeans	14,800,000	285	212	237	4,320,000	4,260,000	15.1	3.9	1.95
tradesoap	8,200,000	159	223	322	2,610,000	2,510,000	78,100	50,800	25,400
xalan	126,000	125	24.2	32.7	11,900	11,500	1	0.0933	0

Table D.3.: Statistics for prm4j and UnsafeMapIterator

			Memory (MB)		Bindings		Monitors		
Benchmark	Events	Runtime (%)	Mean	Max	Created	Collected	Created	Orphaned	Collected
avrora	0	94.9	8.87	8.87	0	0	0	0	0
batik	36,900	100	42.5	57.2	18,700	8,710	3	0	0
eclipse	145,000	97.8	29	29	0	0	0	0	0
fop	1,040,000	205	46	68.1	184,000	145,000	1	0	0
h2	26,500,000	106	221	221	0	0	0	0	0
jython	437,000	312	68.7	77.5	24,200	22,300	2	0	0
luindex	0	101	11.6	11.6	0	0	0	0	0
pmd	8,280,000	221	54.7	70.8	792,000	776,000	1	0	0
sunflow	0	99.1	7.78	7.78	0	0	0	0	0
tomcat	0	115	39.3	39.3	0	0	0	0	0
tradebeans	21,800,000	127	162	162	0	0	0	0	0
tradesoap	9,430,000	113	158	158	0	0	0	0	0
xalan	0	113	15.1	15.1	0	0	0	0	0

Table D.4.: Statistics for prm4j and UnsafeSyncCollection

			Memory (MB)		Bindings		Monitors		
Benchmark	Events	Runtime (%)	Mean	Max	Created	Collected	Created	Orphaned	Collected
avrora	0	95.6	8.82	8.82	0	0	0	0	0
batik	0	98.8	52.7	52.7	0	0	0	0	0
eclipse	153,000	98.6	29	29	0	0	0	0	0
fop	1,040,000	201	45.5	67.2	185,000	145,000	3	0	0
h2	0	104	220	220	0	0	0	0	0
jython	708,000	352	50	50	0	0	0	0	0
luindex	0	102	11.2	11.2	0	0	0	0	0
pmd	8,610,000	221	54.7	72.9	812,000	796,000	1	0	0
sunflow	0	98.3	7.69	7.69	0	0	0	0	0
tomcat	0	115	39.4	39.4	0	0	0	0	0
tradebeans	22,800,000	134	164	164	0	0	0	0	0
tradesoap	10,500,000	114	161	161	0	0	0	0	0
xalan	0	116	15.3	15.3	0	0	0	0	0

Table D.5.: Statistics for prm4j and UnsafeSyncMap

Bibliography

[1] C. Allan, P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. De Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. *ACM SIGPLAN Notices*, 40(10):345–364, 2005.

[2] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *ACM SIGPLAN Notices*, volume 42, pages 589–608. ACM, 2007.

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[4] A. Blass, N. Dershowitz, and Y. Gurevich. When are two algorithms the same? *Bulletin of Symbolic Logic*, 15(2): 145–168, 2009.

[5] J. Bloch. *Effective java*. Prentice Hall, 2008.

[6] E. Bodden. Stateful breakpoints: A practical approach to defining parameterized runtime monitors. In *ESEC/FSE*, volume 11, 2011.

[7] E. Bodden, F. Chen, and G. Roşu. Dependent advice: A general approach to optimizing history-based aspects. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 3–14. ACM, 2009.

[8] E. Bodden. *Verifying finite-state properties of large-scale programs*. PhD thesis, McGill University, June 2009.

[9] E. Bodden. Mopbox: A library approach to runtime verification. In *Runtime Verification*, volume 7186 of *LNCS*, pages 365–369. Springer, 2012.

[10] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. *ECOOP 2007–Object-Oriented Programming*, pages 525–549, 2007.

[11] F. Chen and G. Roşu. Parametric trace slicing and monitoring. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 246–261, 2009.

[12] F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Workshop on Runtime Verification (RV'03)*, volume 89(2) of *ENTCS*, pages 108 – 127, 2003.

[13] F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)*, pages 569–588. ACM press, 2007.

[14] F. Chen, P. O. Meredith, D. Jin, and G. Roşu. Efficient Formalism-Independent Monitoring of Parametric Properties. *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 383–394, November 2009.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Elements of reusable object-oriented design, 1995.

[16] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42 (10):57–76, 2007.

[17] E. Hilsdale and J. Hugunin. Advice weaving in aspectj. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35. ACM, 2004.

[18] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison Wesley, 2006.

[19] D. Jin. *Making Runtime Monitoring of Parametric Properties Practical*. PhD thesis, University of Illinois, 2012.

[20] D. Jin and P. O. Meredith. Garbage Collection for Monitoring Parametric Properties. *ReCALL*, pages 415–424, 2011.

[21] D. Jin and P. O. N. Meredith. Scalable Parametric Runtime Monitoring. (April), 2012.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. *ECOOP 2001—Object-Oriented Programming*, pages 327–354, 2001.

[23] W. Landi et al. Undecidability of static analysis. *ACM letters on programming languages and systems*, 1(4):323–337, 1992.

[24] M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[25] D. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2005.

[26] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[27] P. O. Meredith and G. Roşu. Runtime verification with the rv system. In *Runtime Verification*, pages 136–152. Springer, 2010.

[28] P. O. Meredith. *Efficient, Expressive, and Effective Runtime Verification*. PhD thesis, University of Illinois at Urbana-Champaign, 2012.

[29] P. O. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. *Automated Software Engineering*, 17(2):149–180, February 2010.

[30] E. Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[31] R. Purandare, M. Dwyer, and S. Elbaum. Monitoring finite state properties: algorithmic approaches and their relative strengths. In *Runtime Verification*, pages 381–395. Springer, 2012.

[32] G. Roşu and F. Chen. Semantics and algorithms for parametric monitoring. *Logical Methods in Computer Science*, 8 (1):1–47, 2012.

[33] G. Roşu and F. Chen. Parametric trace slicing and monitoring (technical report). 2008.

[34] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *Software Engineering, IEEE Transactions on*, SE-12(1):157 –171, jan. 1986.

[35] M. Vardi. An automata-theoretic approach to linear temporal logic. *Logics for concurrency*, pages 238–266, 1996.

[36] N. S. Yanofsky. Towards a definition of an algorithm. *Journal of Logic and Computation*, 21(2):253–286, 2011.