Two Generalisations of Roşu and Chen's Trace Slicing Algorithm A

Clemens Ballarin

aicas GmbH
Haid-und-Neu-Straße 18
76131 Karlsruhe, Germany
ballarin@aicas.com

Abstract. Roşu and Chen's trace analysis algorithm identifies activity streams in a monitored application based on data (such as memory locations) and groups events accordingly into slices. It can be generalised to assign several such activity streams to the same slice, even if data is unrelated. This is useful for monitoring scheduling algorithms, which linearise activity streams that are not necessarily related. The algorithm can be generalised further to impose constraints on the generated slices such that, for example, each trace relates a high-priority activity to a low-priority activity. There are no limitations on constraints other than that constraint solvers efficient enough for runtime analysis need to be available.

Keywords: asynchronous events, constraint solving, runtime monitoring, scheduling, trace slicing

1 Introduction

Slicing separates a stream of monitored events into parts, called *slices*, that can be analysed independently of each other. In Roşu and Chen's Algorithm A [11] the separation is based on the data contained in the events. Events that share a piece of data — for example, the address of an object in memory — are identified as related and are put into the same slice. The algorithm is motivated by the observation that activities in a program that operate on separate sets of objects are usually not related. For example, when an iterator is created in a Java program, and the task is to monitor that the underlying collection is not modified while an iterator is used, operations on iterators created from other collections are irrelevant and these events need not (and should not) be put into the slice corresponding to that iterator.

There are monitoring scenarios where it is desirable that events triggered by activities not related directly to each other in the above sense are put into the same slice. An example are scheduling algorithms, which ensure that concurrent activities are executed in an appropriate order — for example, based on priority. Algorithm A is not directly applicable to such scenarios, but it can be extended in a straightforward manner to make it applicable. How this can be done is the subject of the present work.

1.1 Fixed-Priority Scheduling

Fixed-priority scheduling is the most commonly used scheme for scheduling activities in realtime systems [5]. In order to schedule m concurrent activities (for example, threads) on $n < m$ executors (for example, CPUs) each activity is assigned a priority and only the n activities of highest priority are executed. Activities with lower priorities can only make progress when higher-priority activities are blocked.

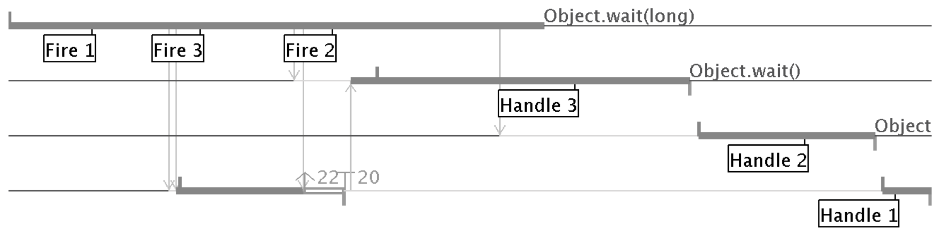

Fig. 1. Firing and Handling of Asynchronous Events on a Multicore System

Fig. 1 shows an execution trace of a Java program running on the realtime-capable JamaicaVM [1] on VxWorks 6.9. The application is based on the Real-Time Specification for Java (RTSJ) [13] and contains threads communicating through *asynchronous events*. The trace was obtained with JamaicaVM's built-in monitoring facilities. It shows four threads running on two CPUs. The first thread, which runs on its own CPU, fires three asynchronous events 1, 2 and 3, and is then suspended in the Java method `wait(long)`. The other three threads share the other CPU and act as handlers. The box labelled "Fire 1" marks the point in time where 1 is fired. A short while later, the corresponding handler, which is the lowest priority thread, is woken up and starts its activity. Concurrently, the first thread now fires 3. This asynchronous event has higher priority than 1 and is handled by the second thread. In order to do so, this thread now apparently needs to enter a lock currently held by the handler of 1, which is of lower priority. This situation is known as *priority inversion*. To prevent a deadlock the priority of the thread holding the lock is temporarily raised. Then, that handler can proceed, the priority is lowered, and 3 is handled. Afterwards, asynchonous event 2, which has been fired in the meantime, is handled, and eventually the handler of 1 resumes and completes its task.

The boxes labelled "Fire x" and "Handle y" in the diagram represent monitor events in the application's execution trace, and they will be abbreviated as $f(x)$ and $h(y)$, where x and y denote asynchronous events.[1] For scheduling to be correct, whenever several asynchronous events are pending simultaneously, the

[1] Events in the execution trace should not be confused with asynchronous events of the RTSJ. When the meaning of "event" is not clear from the context the term "monitor event" will be used for referring to an event in a trace.

higher-priority asynchronous events need to be handled first. That is, for two asynchronous events x and y with priority$(x) >$ priority(y) whenever $f(x)$ is observed then there may either be no $f(y)$ until $h(x)$ or otherwise $h(x)$ must come before $h(y)$.

Such properties can, for example, be expressed with linear temporal logic and monitored with automata [8]. In a scenario with m asynchronous events, rather than constructing a monitor for m asynchronous events, using monitors for pairs of asynchronous events in combination with slicing is more practical.

1.2 Overview of the Paper

Roşu and Chen's algorithm processes a trace of events and computes mappings from event parameters to data. Each mapping yields a slice. The mappings are partial functions and will be called parameter instantiations. In this paper, it will first be shown that the correctness proof of Algorithm A is even valid when generalised from partial functions to semilattices (Sect. 2). Then the algorithm will be extended so it can combine events from multiple activity streams into one slice, and it becomes applicable to monitoring scheduling algorithms (Sect. 3). The extended algorithm yields two slices for each pair of asynchronous events, while according to priorities only one slice is useful. It will then be shown how this can be addressed by generalising the algorithm further with constraint solving techniques (Sect. 4).

2 Algorithm A Revisited

Roşu and Chen's algorithm computes a set of partial functions, which are parameter instantiations, and a slice for each instantiation. In this section the algorithm and its correctness proof are shown to be valid if instantiations are generalised to an arbitrary semilattice. The exposition of semilattices and partial functions follows Jacobson's textbook on basic algebra [7].

2.1 Partial Orders and Upper Semilattices

Lattices are partial orders in which least upper bounds and greatest lower bounds exist. The notions of least upper bound and greatest lower bound are dual, and a lattice can be seen as comprising two semilattices. The semilattice formed by least upper bounds is sufficient for understanding Algorithm A.

Definition 1. *A* partially ordered set *is a tuple* $(S, \leq)$ *where* S *is a set and* $\leq$ *is a binary relation on* S *satisfying reflexivity, antisymmetry and transitivity.*

Let $A \subseteq S$. An element $u \in S$ in an *upper bound* of A if $x \leq u$ for every $x \in A$. It is a *least upper bound* of A if it is an upper bound of A and $u \leq v$ for every upper bound v of A. If a least upper bound exists for A it is unique. The least upper bound of A is denoted as $\bigvee A$. If $\bigvee A \in A$ then the least upper bound of A is also called the *greatest* element of A and denoted $\max A$.

Definition 2. *An* (upper or join) semilattice *is a partially ordered set* $(L, \leq)$ *in which any two elements have a least upper bound.*

The least upper bound of x and y is denoted as $x \vee y$ ("x join y"). By induction, any non-empty finite set of elements of a semilattice has a least upper bound. The least upper bound of $x_1, x_2, \ldots, x_n$ is denoted as $x_1 \vee x_2 \vee \cdots \vee x_n$. A partially ordered set for which every subset A has a least upper bound is called a *complete (upper) semilattice*. The following properties of the join operation of semilattices are generally known to hold for lattices (with both meet and join), but proofs [7, Chap. 8] already apply to semilattices.

Lemma 1. *The join operation* $\vee$ *of a semilattice satisfies commutativity, associativity and idempotence. The order relation and the join operation have these relationships:*

1. $x \leq y$ *if, and only if* $x \vee y = y$.
2. *If* $x \leq z$ *and* $y \leq z$ *then* $x \vee y \leq z$.
3. *If* $x \leq y$ *then* $x \vee z \leq y \vee z$ *(monotonicity).*

For a finite semilattice every non-empty subset A has a least upper bound. Likewise for a complete semilattice. If A coincides with the underlying set L of the semilattice then $\bigvee A$ is the greatest element of L. $\bigvee L$ is called the *top* element of L and denoted $\top$. Conversely, let A be the empty set $\varnothing$. Any $u \in L$ is an upper bound of $\varnothing$. If a least upper bound exists for $\varnothing$ it is called the *bottom* element of L, and $\bigvee \varnothing$ is denoted by $\bot$. Unlike top, not all semilattices have a bottom element. By definition, complete semilattices have a bottom element, and finite semilattices with bottom element are complete.

Definition 3. *A subset* M *of a semilattice* L *is called a* sublattice *(more precisely, an* upper subsemilattice*, but the former will be used throughout for brevity) if it is closed under the operation* $\vee$.

It is evident that M is a semilattice relative to the induced join operation of the sublattice. A sublattice of a complete lattice is complete if it contains a bottom element (which need not coincide with the bottom element of L).

Let $a \in L$ be fixed. The subset of elements $x \in M$ such that $x \leq a$ is either empty or, by Lemma 1.2, a sublattice of M (and of L). We denote this set by $M[a]$. This observation implies

Lemma 2. *If* M *is a sublattice of* L, $a \in L$ *and* $M[a]$ *is non-empty then its least upper bound is an element of* $M[a]$ *— that is,* $\max M[a]$ *exists.*

Additional Results Roşu and Chen lift the binary join operation to sets: for $M, N \subseteq L$ let

$$M \vee N = \{x \vee y \mid x \in M \text{ and } y \in N\}.$$

If M and N are sublattices of L then $M \vee N$ is a sublattice of L as well. If $\bot \in M$ and $\bot \in N$ then $\bot \in M \vee N$. (In fact, this condition holds for any $x \in L$.)

Lemma 3 ([11, Proposition 8.3]). *Let L be a complete semilattice, let Θ be a sublattice of L, $\bot \in \Theta$ and $\theta \in L$, and let $\theta_1, \theta_2 \in \{\theta\} \vee \Theta$ such that $\theta_1 = \bigvee \Theta[\theta_2]$. Then $\theta_1 = \theta_2$.*

Proof. In the original proof it is shown that $\{\theta' \in \Theta \mid \theta_2 = \theta \vee \theta'\}$ has a greatest element q and $q = \theta_1$. It follows that $\theta_2 = \theta \vee \theta_1 = \theta_1$. The original proof applies, which can be shown by step-by-step inspection. □

2.2 Partial Functions

Roşu and Chen's original algorithm operates on partial functions. These do not form a semilattice, but they can be made one by adding an additional element that will be called the "inconsistent function".

Let S and T be non-empty sets. Functions are sets of tuples $(s,t) \in S \times T$. The set of partial functions from S to T is denoted by $S \rightharpoondown T$, the set of (total) functions as $S \to T$. Let be $\alpha \in S \rightharpoondown T$. The set of elements s such that there is a t with $(s,t) \in \alpha$ is the *domain* of α, written $\operatorname{Dom} \alpha$. If α is total, its domain coincides with S. For a partial function, the domain is allowed to be the empty set. In this case, α is called the *empty function* and is denoted as $\bot$.

The subset relation is a partial order on sets. The set of partial functions $S \rightharpoondown T$ is partially ordered by the subset relation as well. Let $\alpha, \beta \in S \rightharpoondown T$. If $\alpha \subseteq \beta$ then $\operatorname{Dom} \alpha \subseteq \operatorname{Dom} \beta$ and α and β agree on $\operatorname{Dom} \alpha$. If additionally $\alpha \subseteq \gamma$ and $\beta \subseteq \gamma$ for some total function γ, then α and β may be viewed as providing partial information towards γ and β being more informative than α. In the sequel, this order relation on partial functions will be denoted as $\sqsubseteq$.

Let α and β again be arbitrary partial functions $\in S \rightharpoondown T$. They are said to be *compatible* if $\alpha(s) = \beta(s)$ for any $s \in \operatorname{Dom} \alpha \cap \operatorname{Dom} \beta$. It is evident that a least upper bound of α and β exists in $S \rightharpoondown T$ if, and only if α and β are compatible. In particular, $S \rightharpoondown T$ is not a semilattice. This can be rectified by introducing the "inconsistent function" $\top$ and declaring $\alpha \sqsubseteq \top$ for any $\alpha \in (S \rightharpoondown T) \cup \{\top\}$. The latter set will be denoted as $S \overset{\top}{\rightharpoondown} T$.

These considerations show

Lemma 4. *$(S \overset{\top}{\rightharpoondown} T, \sqsubseteq)$ is a semilattice with bottom.*

The bottom element is the empty function and the top element is the "inconsistent function". The least upper bound $\alpha \sqcup \beta$ of two partial functions α and β is either $\{(s,t) \mid (s,t) \in \alpha \text{ or } (s,t) \in \beta\}$ if α and β are consistent or, otherwise, $\top$. The least upper bound of a set of partial functions A is denoted as $\bigsqcup A$.

Injective functions will be needed later. The set of injective partial functions will be denoted by $S \rightharpoondown_i T$ and its extension by the inconsistent function as $S \overset{\top}{\rightharpoondown}_i T$. The latter is a sublattice of $S \overset{\top}{\rightharpoondown} T$. Let α and $\beta \in S \overset{\top}{\rightharpoondown}_i T$. The least upper bound of α and β is $\alpha \sqcup \beta$ if that is injective or $\top$ otherwise. It is denoted as $\alpha \sqcup_i \beta$. The least upper bound of $A \subseteq S \overset{\top}{\rightharpoondown}_i T$ is $\bigsqcup_i A$.

2.3 Traces and Slices

A trace is either a sequence of base events or a sequence of events with data. Let $\mathcal{E}$ be the set of base events, X a set of variables and V a set of values (representing the data). Roşu and Chen model events with data as follows. For each $e \in \mathcal{E}$, $X_e \subseteq X$ is the set of *parameters of* e. An *event with data* consists of a base event e and a parameter instantiation $\theta \in X \rightharpoondown V$ such that $\operatorname{Dom}\theta = X_e$. We denote the set of events with data as $\mathcal{E}\langle X \rightharpoondown V\rangle$. The sets $\mathcal{E}^*$ and $\mathcal{E}\langle X \rightharpoondown V\rangle^*$ are the sets of sequences of events; they include the empty sequence ε.

Definition 4. *For a trace $\tau \in \mathcal{E}\langle X \rightharpoondown V\rangle^*$ and a parameter instantiation θ, the* slice $\tau{\restriction_\theta}$ *is the subsequence of base events e such that $e\langle\theta'\rangle \in \tau$ and $\theta' \sqsubseteq \theta$.*

That is, $\tau{\restriction_\theta}$ contains all events of τ whose instantiation is less informative than or equal to θ.

Since a trace is finite, the sets $\mathcal{E}$ and V can be assumed to be finite, and since each base event has only a finite number of parameters, a finite set X of parameters is sufficient as well. Consequently, the set of partial functions $X \rightharpoondown V$ is finite, and $X \overset{\top}{\rightharpoondown} V$ is a complete semilattice. Its bottom element is the empty function $\bot$.

Lemma 5 (Lookup [11, Proposition 14]). *Let τ be a trace with data, and let Θ be a sublattice of $X \overset{\top}{\rightharpoondown} V$ such that $\{\theta' \mid e\langle\theta'\rangle \in \tau\} \subseteq \Theta$ and $\bot \in \Theta$. Let $\theta \in X \overset{\top}{\rightharpoondown} V$. Then $\tau{\restriction_{\bigsqcup \Theta[\theta]}} = \tau{\restriction_\theta}$.*

Proof. The least upper bound $\bigsqcup \Theta[\theta]$ exists by Lemma 2 and is an element of $\Theta[\theta]$. Consider an arbitrary event $e\langle\theta'\rangle \in \tau$. By the premises $\theta' \in \Theta$. It is sufficient to show that $\theta' \sqsubseteq \bigsqcup \Theta[\theta]$ if, and only if $\theta' \sqsubseteq \theta$. Let $\theta' \sqsubseteq \bigsqcup \Theta[\theta]$; θ is an upper bound of $\Theta[\theta]$ and so $\theta' \sqsubseteq \bigsqcup \Theta[\theta] \sqsubseteq \theta$. Conversely, let $\theta' \sqsubseteq \theta$. Then $\theta' \sqsubseteq \theta_0$ for any upper bound θ_0 of $\Theta[\theta]$ and in particular for $\bigsqcup \Theta[\theta]$. □

2.4 The Algorithm

Roşu and Chen's slicing algorithm reads a sequence of events with data and computes sequences of *base events*. The algorithm is shown in Fig. 2. The input is a trace $\tau \in \mathcal{E}\langle X \rightharpoondown V\rangle^*$, which is processed sequentially. The computation yields a set of parameter instantiations $\Theta \subseteq X \overset{\top}{\rightharpoondown} V$ and a map $\mathbb{T} \in (X \overset{\top}{\rightharpoondown} V) \rightharpoondown \mathcal{E}^*$. The latter is the table of slices computed by the algorithm.

Apart from notational details, the only difference to the original version [11, Fig. 2] is the inclusion of the "inconsistent function" $\top$ in Θ and $\operatorname{Dom}\mathbb{T}$. This modification ensures that these sets are semilattices and serves simplifying the correctness argument. Semilattice replaces Roşu and Chen's notion of a closed set of partial functions. Implementations can either exclude $\top$ right away (as done in the original version) or drop the additional trace when returning the result.

Algorithm A

Input $\tau \in \mathcal{E}\langle X \rightharpoonup V\rangle^*$

Output $\mathbb{T} \in (X \overset{\top}{\rightharpoonup} V) \rightharpoonup \mathcal{E}^*$ and $\Theta \subseteq X \overset{\top}{\rightharpoonup} V$

1: $\mathbb{T} \leftarrow \{\bot \mapsto \varepsilon, \top \mapsto \varepsilon\}$; $\Theta \leftarrow \{\bot, \top\}$
2: **for each** $e\langle\theta\rangle \in \tau$ **do**
3:     **for each** $\theta' \in \{\theta\} \sqcup \Theta$ **do**
4:         $\mathbb{T}(\theta') \leftarrow \mathbb{T}(\bigsqcup \Theta[\theta'])e$
5:     **end for**
6:     $\Theta \leftarrow \{\bot, \theta\} \sqcup \Theta$
7: **end for**

Fig. 2. Roşu and Chen's Original Algorithm

Algorithm A with Patterns

Input $P \subseteq A(X)$ and $\tau \in A(V)^*$

Output $\mathbb{T} \in (X \overset{\top}{\rightharpoonup} V) \rightharpoonup A(V)^*$ and $\Theta \subseteq X \overset{\top}{\rightharpoonup} V$

1: $\mathbb{T} \leftarrow \{\bot \mapsto \varepsilon, \top \mapsto \varepsilon\}$; $\Theta \leftarrow \{\bot, \top\}$
2: **for each** $q \in \tau$ **do**
3:     $\Sigma \leftarrow \{\mathrm{mgm}_i(p, q) \mid p \in P\}$
4:     **for each** $\theta' \in \Sigma \sqcup_i \Theta$ **do**
5:         $\mathbb{T}(\theta') \leftarrow \mathbb{T}(\bigsqcup_i \Theta[\theta'])q$
6:     **end for**
7:     $\Theta \leftarrow \Theta \cup (\Sigma \sqcup_i \Theta)$
8: **end for**

Fig. 3. Data Interpretation Based on Patterns

Algorithm A with Constraints

Input $c \in C$, $P \subseteq A(X)$ and $\tau \in A(V)^*$

Output $\mathbb{T} \in C \rightharpoonup A(V)^*$ and $\Theta \subseteq C$

1: $\mathbb{T} \leftarrow \{c \mapsto \varepsilon, \top \mapsto \varepsilon\}$; $\Theta \leftarrow \{c, \top\}$
2: **for each** $q \in \tau$ **do**
3:     $\Sigma \leftarrow \{\mathit{true}.\mathrm{mgm}(p, q) \mid p \in P\}$
4:     **for each** $\theta' \in \Sigma \wedge \Theta$ **do**
5:         $\mathbb{T}(\theta') \leftarrow \mathbb{T}(\bigwedge \Theta[\theta'])q$
6:     **end for**
7:     $\Theta \leftarrow \Theta \cup (\Sigma \wedge \Theta)$
8: **end for**

Fig. 4. Trace Slicing with Constraints

Theorem 1 (Slicing [11, Theorem 1]). *Let* $\tau \in \mathcal{E}\langle X \rightharpoonup V\rangle^*$ *be a trace where* $\mathcal{E}$*,* X *and* V *are sets of events, parameters and values, respectively, and let* $\mathbb{T}$ *and* Θ *be the result of processing* τ *with Algorithm A. Then these conditions hold:*

$$\operatorname{Dom}\mathbb{T} = \Theta \tag{1}$$

$$\{\theta \mid e\langle\theta\rangle \in \tau\} \subseteq \Theta \tag{2}$$

$$\mathbb{T}(\theta) = \tau{\restriction}_\theta \text{ for any } \theta \in \Theta \tag{3}$$

$$\tau{\restriction}_\theta = \mathbb{T}(\bigsqcup \Theta[\theta]) \text{ for any } \theta \in X \overset{\top}{\rightharpoonup} V \tag{4}$$

The proof follows Roşu and Chen's proof.

Proof. All arguments are by induction on the outer loop. Let $\mathbb{T}$ and Θ denote the states of the variables $\mathbb{T}$ and Θ at the beginning of the body of the outer loop and $\mathbb{T}'$ and Θ' the states at the end, and let $e\langle\theta\rangle$ be the processed event.

First, Θ is a semilattice with bottom element $\bot$: the set is initialised to $\{\bot, \top\}$, which is a sublattice of $X \overset{\top}{\rightharpoonup} V$, in line 1, and it remains one when updated to $\{\bot, \theta\} \sqcup \Theta$ in line 6 since the join operation on sets preserves sublattices. Moreover, Θ is complete.

Equation (1) is also immediate from how $\mathbb{T}$ and Θ are updated. In particular, the inner loop defines $\mathbb{T}$ at $\{\theta\} \sqcup \Theta$ (if not defined already) and so

$$\operatorname{Dom}\mathbb{T}' = \operatorname{Dom}\mathbb{T} \cup \{\theta\} \sqcup \Theta = \Theta \cup \{\theta\} \sqcup \Theta = \{\bot, \theta\} \sqcup \Theta = \Theta'$$

by the induction hypothesis and the definition of the join operator on sets.

Condition (2) is again immediate from the updates in lines 1 and 6.

The sequence in which the elements of $\{\theta\} \sqcup \Theta$ are processed by the inner loop (lines 3 to 5) is not specified, and in particular an event e must not be added to the same slice twice. In fact, the outcome of the loop is invariant under the processing sequence, and it is sufficient to show that the order of two elements $\theta_1, \theta_2 \in \{\theta\} \sqcup \Theta$ that are processed consecutively in the loop does not matter. This is a consequence of Lemma 3.

For (3) the induction hypothesis is $\mathbb{T}(\theta_0) = \tau{\restriction}_{\theta_0}$ for any $\theta_0 \in \Theta$. Let $\theta' \in \Theta'$. The event e is added to the slice $\mathbb{T}(\theta')$ if, and only if $\theta' \in \{\theta\} \sqcup \Theta$. This is equivalent to $\theta \sqsubseteq \theta'$ since $\theta' \in \Theta' = \Theta \cup (\{\theta\} \sqcup \Theta)$, and the assignment in line 4 updates the correct slot. It remains to be shown that, if e is added to a slot, the table lookup retrieves the correct prefix: $\mathbb{T}(\bigsqcup \Theta[\theta']) = \tau{\restriction}_{\bigsqcup \Theta[\theta']} = \tau{\restriction}_{\theta'}$. This follows from the induction hypothesis and Lemma 5.

Equation (4) follows from (3) and Lemma 5. □

3 Combining Multiple Activity Streams into One Slice

Let us now return to the monitoring problem of Sect. 1.1, where asynchronous events are fired and handled in an application. Firing and handling are traced as $f(x)$ and $h(x)$, respectively. In order to monitor whether for each pair of asynchronous events x and y the order of the recorded fire and handle monitor

events adheres to the scheduling policy, slices for each pair of events are extracted from the trace. Consider, for example,

$$\tau = f(2)h(2)f(1)f(2)h(2)f(3)h(3)h(1)$$

with three asynchronous events 1, 2 and 3, and priority(3) $>$ priority(2) $>$ priority(1). Slicing should yield a trace for each pair of asynchronous events:

$$\begin{aligned}\tau_{2,1} &= f(2)h(2)f(1)f(2)h(2)h(1)\\ \tau_{3,1} &= f(1)f(3)h(3)h(1)\\ \tau_{3,2} &= f(2)h(2)f(2)h(2)f(3)h(3)\end{aligned}$$

Each slice can then be processed individually — for example, by an instance of a suitable parametric automaton where x represents the high-priority and y the low-priority asynchronous event.

This example illustrates the main challenge when combining multiple activity streams into one slice. Distinct instances of $f(x)$ and $f(y)$ need to be put in the same slice. Algorithm A is not designed to support this. In fact, this is a limitation of the trace model.

3.1 Events and Event Patterns

The modified trace model uses terminology from term algebra [2], but it is only a subtle generalisation of the original model. Like in Sect. 2.3, X and V denote the sets of parameters (variables) and values, respectively, of the base events. A *term* is either a parameter or a value. $T = X \cup V$ denotes the set of terms. Base events are now symbols that can be applied to a fixed number of terms. Let $e \in \mathcal{E}$. Then $\alpha_e \in \mathbb{N}$ is the *arity* of e. Let $\alpha_e = k$ and $t_1, \ldots, t_k \in T$. Then $e(t_1, \ldots, t_k)$ is an *atom*; if $k = 0$ then the atom is denoted as e. If $t_1, \ldots, t_k \in V$ then $e(t_1, \ldots, t_k)$ is a *ground atom*. The set of atoms is denoted as $A(X, V)$, the set of ground atoms as $A(V)$. Atoms are also called *patterns*, and ground atoms now represent events with data. $A(X)$ is the set of patterns that contain no values.

An instantiation is again a partial function $\theta \in X \rightharpoonup V$. The result of its application to a term t is denoted as $\theta(t)$ and is defined as $\theta(t)$ if $t \in \operatorname{Dom}\theta$. Otherwise, it is t. In particular, a value is mapped to itself. Instantiations are lifted to atoms: $\sigma(e(t_1, \ldots, t_k)) = e(\sigma(t_1), \ldots, \sigma(t_k))$.

Instantiations are no longer considered part of the trace but are inferred by matching events against patterns. A pattern $p \in A(X, V)$ *matches* an event $q \in A(V)$ if there is an instantiation $\theta \in X \rightharpoonup V$ such that $\theta(p) = q$, and θ is a *matcher* of p and q. A minimal (or most general) matcher of p and q maps exactly the variables that occur in p to the corresponding values in q. If a matcher exists for p and q the minimal matcher is unique. It is denoted as $\mathrm{mgm}(p, q)$. We define $\mathrm{mgm}(p, q) = \top$ if p does not match q. If the minimal matcher is injective, $\mathrm{mgm}_i(p, q)$ is defined as $\mathrm{mgm}(p, q)$. Otherwise $\mathrm{mgm}_i(p, q) = \top$.

3.2 Slicing Based on Patterns

A slice of a trace $\tau \in A(V)^*$ is a subsequence of events that match an element of a given set $P \subseteq A(X)$ of patterns.

Definition 5 (Slice with Patterns). *For a set of patterns $P \subseteq A(X)$, a trace $\tau \in A(V)$ and a parameter instantiation θ, the slice $\tau|_\theta$ is defined as follows. Either $\theta = \top$. Then $\tau|_\theta$ is the full sequence τ. Otherwise, it is the subsequence of events $q \in \tau$ such that there exists a pattern $p \in P$ and $\theta(p) = q$.*

The modified slicing algorithm is shown in Fig. 3. It takes the set P of patterns as an additional argument. The inner loop iterates over parameter instantiations θ' that extend minimal matchers of these patterns and the processed event q. Only instantiations that are injective are considered.

Lemma 6 (Lookup with Patterns). *Let $P \subseteq A(X)$ and $\tau \in A(V)^*$, and let Θ be a sublattice of $X \xrightarrow{\top}_i V$ such that $\{\mathrm{mgm}_i(p,q) \mid p \in P,\, q \in \tau\} \subseteq \Theta$ and $\bot, \top \in \Theta$. Let $\theta \in X \xrightarrow{\top}_i V$. Then $\tau|_{\bigsqcup_i \Theta[\theta]} = \tau|_\theta$.*

Proof. Let $q \in \tau|_\theta$. Either $\theta = \top$, and so $\bigsqcup_i \Theta[\theta] = \top$ and $q \in \tau|_{\bigsqcup_i \Theta[\theta]}$. Otherwise there is a $p \in P$ with $\theta(p) = q$. Then $\mathrm{mgm}_i(p,q) \sqsubseteq \theta$ and $\mathrm{mgm}_i(p,q) \in \Theta$. Therefore $\mathrm{mgm}_i(p,q) \sqsubseteq \bigsqcup_i \Theta[\theta]$ and $(\bigsqcup_i \Theta[\theta])(p) = q$. This implies $q \in \tau|_{\bigsqcup_i \Theta[\theta]}$. Conversely, let $q \in \bigsqcup_i \Theta[\theta]$. Either $\bigsqcup_i \Theta[\theta] = \top$ and so $\theta = \top$ or there is a $p \in P$ such that $(\bigsqcup_i \Theta[\theta])(p) = q$. From $\bigsqcup_i \Theta[\theta] \sqsubseteq \theta$ follows $\theta(p) = q$ and $q \in \tau|_\theta$.

Theorem 2 (Slicing with Patterns). *Let $P \subseteq A(X)$ and $\tau \in A(V)^*$, and let $\mathbb{T}$ and Θ be the result of processing P and τ with Algorithm A with Patterns. Then these conditions hold:*

$$\mathrm{Dom}\,\mathbb{T} = \Theta \tag{5}$$

$$\{\mathrm{mgm}_i(p,q) \mid p \in P \text{ and } q \in \tau\} \subseteq \Theta \tag{6}$$

$$\mathbb{T}(\theta) = \tau|_\theta \text{ for any } \theta \in \Theta \tag{7}$$

$$\tau|_\theta = \mathbb{T}(\bigsqcup\nolimits_i \Theta[\theta]) \text{ for any } \theta \in X \xrightarrow{\top}_i V \tag{8}$$

The conditions are direct analogues of those of Theorem 1 except (6), which says that traces for all minimal matchers of patterns and events are computed.

Proof. The argument that Θ is a semilattice is more involved than for Theorem 1 since a set of matchers is processed in each iteration of the outer loop. It is easy to see that $\Theta \subseteq X \xrightarrow{\top}_i V$ throughout the computation. Further, in line 7, Θ is updated to $\Theta \cup (\Sigma \sqcup_i \Theta) = (\{\bot, \top\} \cup \Sigma) \sqcup_i \Theta$. The set $\{\bot, \top\} \cup \Sigma$ is a semilattice where the elements of Σ are not comparable. Let σ_1 and σ_2 be two distinct elements of Σ. They are matchers of distinct patterns with the same event q. Since these patterns only contain variables, $\sigma_1 \sqcup \sigma_2$ is not injective, and so $\sigma_1 \sqcup_i \sigma_2 = \top$. By induction, Θ is a sublattice of $X \xrightarrow{\top}_i V$.

Since the proof of Theorem 1 applies to semilattices in general, most of the reasoning is directly applicable to slicing with patterns. The exception is (7)

because the definition of slice has changed. The induction hypothesis is $\mathbb{T}(\theta_0) = \tau|_{\theta_0}$ for any $\theta_0 \in \Theta$, and let q again be the processed event. Consider $\theta' \in \Theta' = \Theta \cup (\Sigma \sqcup_i \Theta)$ where $\Sigma = \{\mathrm{mgm}_i(p, q) \mid p \in P\}$. The event q is added to the slice $\mathbb{T}(\theta')$ if, and only if $\theta' \in \Sigma \sqcup_i \Theta$. There are two cases. Either $\theta' \in \Sigma \sqcup_i \Theta$. Then there is a pattern $p \in P$ and $\theta' \in \{\mathrm{mgm}_i(p, q)\} \sqcup_i \Theta$, and so $\theta'(p) = q$ and $q \in \tau q|_{\theta'}$. Otherwise, $\theta' \notin \Sigma \sqcup_i \Theta$ but $\theta' \in \Theta$. If there were a $p \in P$ such that $\theta'(p) = q$ then, by the minimality of the elements of Σ, $\theta' \in \Sigma \sqcup_i \Theta$. So $q \notin \tau q|_{\theta'}$. Therefore the assignment in line 5 updates the correct slot. By the induction hypothesis and Lemma 6 the table lookup in the same line retrieves the correct prefix: $\mathbb{T}(\bigsqcup_i \Theta[\theta']) = \tau|_{\bigsqcup_i \Theta[\theta']} = \tau|_{\theta'}$. □

3.3 Examples

It is illustrative to inspect the working of the algorithm. We return to the scheduling example from the beginning of the section. Let $P = \{f(x), f(y), h(x), h(y)\}$ and let

$$\tau = f(2)h(2)f(1)f(2)h(2)f(3)h(3)h(1)$$

be the monitored trace. After processing the first two events, the set of instantiations Θ is $\{\bot, \{x \mapsto 2\}, \{y \mapsto 2\}, \top\}$ and $\mathbb{T}$ contains four slices:

$$\mathbb{T}(\bot) = \varepsilon, \quad \mathbb{T}(\{x \mapsto 2\}) = \mathbb{T}(\{y \mapsto 2\}) = \mathbb{T}(\top) = f(2)h(2)$$

When the next event, $f(1)$, is processed the set of instantiations is duplicated, and $f(1)$ is added to all slices of instantiations that contain a mapping to 1 (and of $\top$). The full result of processing τ is shown in Table 1. The third group of slices, containing mappings to 3, is added when $f(3)$ is processed. One can see that indeed the expected slices are computed. For example, $\tau_{2,1} = \mathbb{T}(\{x \mapsto 2, y \mapsto 1\})$.

Slices for instantiations that map only one parameter to a value, for example for $\{x \mapsto 2\}$, appear to be redundant but are required so they can be cloned whenever a monitor event for a new asynchronous event arrives. On the other hand, having two instantiations for each pair of asynchronous events, such as $\{x \mapsto 1, y \mapsto 2\}$ in addition to $\{x \mapsto 2, y \mapsto 1\}$ is redundant. It is desirable to only compute slices for instantiations with $\mathrm{priority}(x) > \mathrm{priority}(y)$. This will be the subject of the following section.

A trace processed by the original Algorithm A is also amenable for processing by Algorithm A with Patterns. For each base event $e \in \mathcal{E}$ let the set P of patterns contain one $f(x_1, \ldots, x_k)$ such that $X_e = \{x_1, \ldots, x_k\}$ and $x_1, \ldots, x_k$ is the intended order of parameters. Since each event q is matched by exactly one pattern $p \in P$ the set of instantiations Σ computed in line 3 is singleton and the algorithm is schematically reduced to the original version. The remaining difference is that the former operates on parameter instantiations that are injective. The author believes that this is not a fundamental limitation. It appears that in typical applications of the original algorithm, only injective parameter instantiations are relevant.

θ	$\mathbb{T}(\theta)$							
$\bot$	ε							
$x \mapsto 2$	$f(2)$	$h(2)$		$f(2)$	$h(2)$			
$y \mapsto 2$	$f(2)$	$h(2)$		$f(2)$	$h(2)$			
$\top$	$f(2)$	$h(2)$	$f(1)$	$f(2)$	$h(2)$	$f(3)$	$h(3)$	$h(1)$
$x \mapsto 1$			$f(1)$					$h(1)$
$y \mapsto 1$			$f(1)$					$h(1)$
$x \mapsto 1, y \mapsto 2$	$f(2)$	$h(2)$	$f(1)$	$f(2)$	$h(2)$			$h(1)$
$x \mapsto 2, y \mapsto 1$	$f(2)$	$h(2)$	$f(1)$	$f(2)$	$h(2)$			$h(1)$
$y \mapsto 3$						$f(3)$	$h(3)$	
$x \mapsto 3$						$f(3)$	$h(3)$	
$x \mapsto 1, y \mapsto 3$			$f(1)$			$f(3)$	$h(3)$	$h(1)$
$x \mapsto 2, y \mapsto 3$	$f(2)$	$h(2)$		$f(2)$	$h(2)$	$f(3)$	$h(3)$	
$x \mapsto 3, y \mapsto 1$			$f(1)$			$f(3)$	$h(3)$	$h(1)$
$x \mapsto 3, y \mapsto 2$	$f(2)$	$h(2)$		$f(2)$	$h(2)$	$f(3)$	$h(3)$	

Table 1. Slices for $\tau = f(2)h(2)f(1)f(2)h(2)f(3)h(3)h(1)$

4 Adding Constraints

The slicing algorithm with patterns computes slices for all combinations of asynchronous events (see Table 1), while only slices for events with $x > y$ or, more precisely, priority$(x) >$ priority(y) are meaningful for the subsequent analysis. It is easy to filter out the undesired slices before passing them on to analysis. But it is even possible to avoid generating these slices right away by imposing constraints on the parameter instantiation.

4.1 Constraint Solving

Constraint solving is an established technique from the problem solving domain, and efficient solvers exist for many arithmetic and finite domains. The following introduction follows Marriott and Stuckey's textbook [10].

A *constraint domain* defines the language of constraints and the set of values over which they range. Constraints involve variables. The set of variables and values are X and V, respectively, sharing the notation for instances from the previous sections. A *constraint* c is of the form $\gamma_1 \wedge \cdots \wedge \gamma_n$, where the γ_i are *primitive constraints*. The latter are predicate symbols applied to expressions. The set of expressions is defined by the constraint domain. It includes variables and values.

A *valuation* maps variables to values (the notion is identical to the that of instantiations from the previous section). If a constraint evaluates to *true* under a valuation it is said to have a *solution*. The empty conjunction of primitive constraints, for which any valuation is a solution, is denoted by *true*, the constraint that has no solution as *false*.

Let c_1 and c_2 be constraints. The conjunction $c_1 \wedge c_2$ is the conjunction of the primitive constraints of c_1 and c_2. The constraint c_1 *implies* c_2 if $\theta(c_1) = \mathit{true}$ implies $\theta(c_2) = \mathit{true}$ for all valuations θ such that $\mathrm{Dom}\,\theta$ contains the variables that occur in c_1 and c_2. Implication is denoted by $c_1 \to c_2$. If $c_1 \to c_2$ and $c_2 \to c_1$ then c_1 and c_2 are *equivalent.* In the sequel, equivalent constraints are considered equal. It is also assumed that equations of the form $x = v$ for $x \in X$ and $v \in V$ can be expressed as constraints, and that implication and equivalence are computable — that is, a complete solver exists. For $\theta \in X \rightharpoonup V$ we denote the extension of a constraint c by the equational constraints given by the mappings of θ as $c.\theta$. For the "inconsistent function", $c.\top = \mathit{false}$.

The set C of constraints for a constraint domain is partially ordered by implication. It is also partially ordered by reverse implication. The latter is an upper semilattice where $c_1 \wedge c_2$ is the least upper bound of c_1 and c_2. The least upper bound of a set $C' \subseteq C$ of constraints is denoted by $\bigwedge C'$. The top element is *false* and the bottom element *true*.[2]

4.2 Slicing with Constraints

Slicing with constraints involves an *initial constraint* c, which is taken into account while matching patterns to events. For example, the constraints $x > y$ (or $\mathrm{priority}(x) > \mathrm{priority}(y)$) expresses the priority condition for the scheduling example, where $\{f(x), f(y), h(x), h(y)\}$ is the set of patterns. This constraint already entails injectivity of parameter instantiations. Explicit references to injectivity can be pulled out of the algorithm by requiring the initial constraint to imply injectivity of instantiations.

Definition 6 (Slice with Constraints). *For an initial constraint $c \in C$, a set of patterns $P \subseteq A(X)$, a trace $\tau \in A(V)$ and a constraint θ, the slice $\tau|_\theta$ is defined as follows. Either $\theta = \mathit{false}$. Then $\tau|_\theta$ is the full sequence τ. Otherwise, it is the subsequence of events $q \in \tau$ such that there exists a pattern $p \in P$ and $\theta \to c.\mathrm{mgm}(p, q)$.*

The modified slicing algorithm is shown in Fig. 4. It takes the initial constraint c as an additional argument. The inner loop iterates over constraints θ' that imply c.

Lemma 7 (Lookup with Constraints). *Let $\tau \in A(V)^*$, $c \in C$ and let Θ be a sublattice of C such that $\{c.\mathrm{mgm}(p, q) \mid p \in P,\, q \in \tau\} \subseteq \Theta$ and $c, \top \in \Theta$. Let $\theta \in C$ with $\theta \to c$. Then $\tau|_{\bigsqcup \Theta[\theta]} = \tau|_\theta$.*

Proof. Let $q \in \tau|_\theta$. Either $\theta = \mathit{false}$, and so $\bigwedge \Theta[\theta] = \mathit{false}$ and $q \in \tau|_{\bigwedge \Theta[\theta]}$. Otherwise there is a $p \in P$ with $\theta \to c.\mathrm{mgm}(p, q)$ and $c.\mathrm{mgm}(p, q) \in \Theta$, and so $\bigwedge \Theta[\theta] \to c.\mathrm{mgm}(p, q)$. This implies $q \in \tau|_{\bigwedge \Theta[\theta]}$. Conversely, let $q \in \bigwedge \Theta[\theta]$. Either $\bigwedge \Theta[\theta] = \mathit{false}$ and so $\theta = \mathit{false}$ or there is a $p \in P$ such that $\bigwedge \Theta[\theta] \to c.\mathrm{mgm}(p, q)$. From $\theta \to \bigwedge \Theta[\theta]$ follows $\theta \to c.\mathrm{mgm}(p, q)$ and $q \in \tau|_\theta$.

[2] Logic-inclined readers might find it more intuitive to interpret C as a lower semilattice with top *true* and bottom *false*. For consistency with the previous versions of the algorithm this has not been done here.

Definition and lemma differ from their counterparts from Sect. 3 only in notation, and the proof of the lemma is a literal translation of the version for patterns.

Theorem 3 (Slicing with Constraints). *Let $c \in C$ and $P \subseteq A(X)$ such that $c \wedge (x = y) \to false$ for any two distinct variables $x, y \in X$. Let $\tau \in A(V)^*$ and let $\mathbb{T}$ and Θ be the result of processing τ with Algorithm A with Constraints. Then these conditions hold:*

$$\operatorname{Dom} \mathbb{T} = \Theta \tag{9}$$

$$\{c.\mathrm{mgm}(p, q) \mid p \in P \textit{ and } q \in \tau\} \subseteq \Theta \tag{10}$$

$$\mathbb{T}(\theta) = \tau|_\theta \textit{ for any } \theta \in \Theta \tag{11}$$

$$\tau|_\theta = \mathbb{T}(\bigwedge \Theta[\theta]) \textit{ for any } \theta \in C \textit{ with } \theta \to c \tag{12}$$

Proof. It is sufficient to observe that Θ is maintained during the computation to be a sublattice of C with bottom element c and top element *false*. Then the arguments for Theorem 2 apply. □

4.3 Experiments

Algorithm A with Constraints was implemented in Java. The implementation is generic in the constraint domain and enables experiments with different constraint solvers. It can read traces generated by JamaicaVM, but it is not designed for online monitoring. The implementation is a fairly direct rendering of the pseudocode from Fig. 4 and has not been optimised in any way.

Two solvers were provided. The first is a wrapper for the semilattice $X \xrightarrow{\top}_i V$. The second supports the primitive constraints $x \leq y$ and $x \neq y$. The order constraints are maintained as a directed graph, and whenever there is a $x \neq y$ such that x and y are in the same strongly connected component the collection of primitive constraints has no solution (is inconsistent).

Configuring Algorithm A with Constraints with the solver for injective functions and the initial constraint $c = true$ yields Algorithm A with Patterns. Hence the former is a generalisation of the latter. The slices shown in Table 1 were obtained with this configuration of the implementation. When switching to the solver for order constraints and inequality, and with the initial constraint $x > y$, the result is the same, except that the slices for $\{x \mapsto 1, y \mapsto 2\}$, $\{x \mapsto 1, y \mapsto 3\}$ and $\{x \mapsto 2, y \mapsto 3\}$ disappear as expected. The implementation was tested on a number of traces including the example from Roşu and Chen [11, Table 1] and yielded the expected results.

5 Conclusions

This work was inspired by work by Boden and Stolz [4, 12]. The idea pursued initially was to use a combination of alternating automata and constraint solvers for imposing priority constraints on the monitors. While this seemed to work in

principle, the solution was fairly complicated, and an extension to time — for example, timed automata [9] —, which is indispensable for monitoring realtime systems, would have complicated things even further. The combination of constraint solvers with a slicing algorithm is much cleaner. It enables using any monitoring technology.

An extension to the Algorithms B and C, which are versions of A optimised for use in practice, has not yet been investigated. Neither have we experimented with efficient finite-domain solvers, which seem most appropriate for handling priority constraints. The main requirement for efficiency, namely that the table $\mathbb{T}$ of slices can be updated in parallel in the inner loop of the algorithm, is also met by the two variants of A presented here.

Solutions to monitoring events with data differ in the trade-off between monitoring speed and expressiveness. Quantified event automata [3] share some features of this work — for example, the use of pattern matching to categorise events — but go beyond beyond its specification capabilities by allowing existential quantification of data. Both extensions can monitor properties not amenable to the original algorithm. Temporal data logic [6] is a monitoring formalism based on the combination of a temporal logic for specifying the order of events with a logic for reasoning about the data. Like with our constraint-based extension of Roşu and Chen's algorithm a reasoning specialist — in this case an SMT solver — is used for analysing the data.

The distinctive contribution of this work is that Roşu and Chen's algorithm itself is applicable to a wider range of monitoring problems than it was designed for. Whether an integration of the algorithm with constraint solvers performs better in practice than filtering out unwanted slices generated by the version for patterns remains to be seen. The main result is theoretical: a deeper understanding of Roşu and Chen's algorithm, which opens the field for new applications.

Acknowledgements The work was funded in part by the European Union within the 7th Framework Programme, Project JUNIPER, Grant Agreement 318763, and as Artemis Joint Undertaking, Project CONCERTO, Grant Agreement 333053. Quantified event automata were brought to the attention of the author by two of the anonymous reviewers.

References

1. aicas GmbH, Karlsruhe, Germany: JamaicaVM 6.3 User Manual (2014)
2. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
3. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified event automata: Towards expressive and efficient runtime monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods, Paris, France, pp. 68–84. LNCS 7436, Springer (2012)
4. Bodden, E.: J-LO A tool for runtime-checking of temporal assertions. Diplomarbeit, RWTH Aachen (2005)
5. Burns, A., Wellings, A.: Real-Time Systems and Programming Languages. Addison Wesley, third edn. (2001)

6. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Grenoble, France, pp. 341–356. LNCS 8413, Springer (2014)
7. Jacobson, N.: Basic Algebra I. Freeman, 2nd edn. (1985)
8. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming 78(5), 293–303 (2009)
9. Maler, O., Nickovic, D., Pnueli, A.: From MITL to timed automata. In: Asarin, E., Bouyer, P. (eds.) Formal Modeling and Analysis of Timed Systems, pp. 274–289. LNCS 4202, Springer Berlin Heidelberg (2006)
10. Marriott, K., Stuckey, P.J.: Programming with Constraints. MIT Press (1998)
11. Roşu, G., Chen, F.: Semantics and algorithms for parametric monitoring. Logical Methods in Computer Science 8(1:9), 1–47 (2012)
12. Stolz, V.: Temporal assertions for sequential and concurrent programs. Ph.D. thesis, RWTH Aachen (2006), also Technical Report AIB-2007-15.
13. Wellings, A.: Concurrent and Real-Time Programming in Java. Wiley (2004)