Runtime monitoring for concurrent systems

Yoriyuki Yamagata[1], Cyrille Artho[1,2], Masami Hagiya[3], Jun Inoue[1], Lei Ma[4], Yoshinori Tanabe[5], and Mitsuharu Yamamoto[4]

[1] National Institute of Advanced Industrial Science and Technology (AIST), 1-8-31 Midorigaoka, Ikeda, Osaka 563-8577 Japan {yoriyuki.yamagata, c.artho, jun.inoue}@aist.go.jp
[2] KTH Royal Institute of Technology, 100 44 Stockholm, Sweden artho@kth.se
[3] The University of Tokyo, 7-3-1 Bunkyoku, Tokyo 113-8654 Japan hagiya@is.s.u-tokyo.ac.jp
[4] Chiba University, 1-33 Yayoicho, Inageku, Chiba, Chiba 263-0022 Japan malei@chiba-u.jp, mituharu@math.s.chiba-u.ac.jp
[5] Tsurumi University, 2-1-3 Tsurumi, Tsurumi, Yokohama, Kanagawa 230-0063 Japan tanabe-y@tsurumi-u.ac.jp

Abstract. Most existing specification languages for runtime verification describe the properties of the entire system in a top-down manner, and lack constructs to describe concurrency in the specification directly. CSP_E is a runtime-monitoring framework based on Hoare's Communicating Sequential Processes (CSP) that captures concurrency in the specification directly. In this paper, we define the syntax of CSP_E and its formal semantics. In comparison to quantified event automata (QEA), as an example, CSP_E describes a specification for a concurrent system in a bottom-up manner, whereas QEA lends itself to a top-down manner. We also present an implementation of CSP_E, which supports full CSP_E without optimization. When comparing its performance to that of QEA, our implementation of CSP_E requires slightly more than twice the time required by QEA; we consider this overhead to be acceptable. Finally, we introduce a tool named `stracematch`, which is developed using CSP_E. It monitors system calls in (Mac) OS X and verifies the usage of file descriptors by a monitored process.

Keywords: runtime monitoring, parametric monitoring, CSP, process algebra

1 Introduction

Runtime monitoring is a technique for monitoring program execution. In a subfield of run-time monitoring called specification-based monitoring [36], program execution is monitored against a given specification. The specification is given by a formal language, such as temporal logic, automata, grammar, or rule-based systems. Thus, specification-based monitoring lies somewhere between tests—which usually test only input/output contracts or assertions—and traditional formal methods such as model checking or theorem proving, which try to prove the correctness of a program along all possible execution paths.

This work focuses on *parametric monitoring*. In parametric monitoring, program execution consists of a sequence of events, and each event is associated with one or more parameters, such as a file name or an IP address.

Many frameworks for parametric monitoring have been proposed. One group of frameworks is based on automata. Another is based on formal languages such as regular expressions or context-free grammars, and still others are based on temporal logic or rule-based systems. However, most of these frameworks do not allow for a direct description of the concurrency of a monitored program. Because a bug related to concurrency often surfaces only at runtime, runtime monitoring of a concurrent system is important. In this paper, we investigate the following research questions.

1. Can we design a monitoring language that specifies a concurrent system in a bottom-up fashion—in a way that first describes the specifications of components and then combines them?
2. Can process calculi, which are studied extensively to specify concurrent systems, be used for this particular purpose?

The first question is important because a concurrent system is often specified by the specifications of its components and how they interact. On the other hand, the traditional monitoring languages listed above concentrate on specifying the global properties of systems, regardless of the specifications of their components. This is useful, for instance, when we want to specify a safety property for the whole system, regardless of its components. However, it would be difficult to describe the correct interaction of the components in the system using such methods, because they lack constructs to compose the specifications of systems based on the specifications of their components.

The second question addresses using process calculi in a new context. Process calculi were developed to reason about concurrent systems. In process calculi, programs are processes, which are composed of simpler processes using sequential or concurrent composition. Processes which compose a program can communicate with each other using communication primitives such as events or channels. In particular, we focus on Hoare's Communicating Sequential Processes (CSP) [37, 47]. CSP is a process calculus that uses events as communication primitives. Thus, CSP is effective for describing event sequences generated by a concurrent system.

In this paper, we introduce CSP_E, a runtime-monitoring framework based on CSP. Because it is based on CSP, CSP_E has constructs that allow for the direct description of concurrency in the specification. In this paper, we define the language of CSP_E and its formal semantics. However, our goal is not to introduce a completely new language. Rather, our goal is to show that, with a slight modification, CSP is amenable to monitoring. In fact, to obtain CSP_E from CSP, we merely add a `Failure` constant to express already-failed processes. We show that CSP_E can neatly describe a system of Unix-like processes and file descriptors. We also implement a domain-specific language (DSL) based on CSP_E and compare its performance with quantified event automata (QEA) [5, 36, 46, 45] implemented in [14], using a simulated event log of the Unix-like processes

and file descriptors. QEA is an automata-based monitoring framework, which, according to [38], is one of the most expressive of this kind. Further, it is easiest to understand by engineers [38]. Finally, MarQ, an implementation of QEA, came top in the offline and Java track in first international competition on Runtime Verification (2014) [22]. We use an implementation [14] that is a successor of MarQ. In this comparison, our CSP_E implementation requires slightly more than twice the time that QEA takes; we consider this overhead to be acceptable.

Our DSL is shallowly embedded in the Scala programming language [43]; this language is also named CSP_E. An event monitor can be defined by using the CSP_E language. A monitor that is defined by CSP_E sequentially consumes the events from an event stream. If at some point the monitor fails to proceed, this indicates that something unexpected has occurred.

The language of CSP_E was discussed in part in [3]. In this paper, we fully explain the syntax, semantics, and implementation of CSP_E. In [3], the `process` construct is used for a recursive definition. In this paper, we use the built-in `def` construct in Scala for a recursive definition. If a `process` construct is needed, it can be derived from the `def` construct.

We develop denotational semantics for CSP_E, based on the trace semantics of CSP. Our technical contribution to formal semantics is a proof that the semantic space of CSP_E can still be defined as a complete partial order (CPO) — a mathematical structure which allows a recursive definition.

This paper is organized as follows. In Section 2, we discuss related work. In Section 3, we discuss the syntax of CSP_E using a motivating example. In Section 4, we discuss the formal semantics of CSP_E and prove that it forms a complete partial order. This makes it possible to define monitors recursively. In Section 5, we discuss our implementation of CSP_E. In Section 6, we show a benchmark for our CSP_E implementation and a QEA implementation [14] using the motivating example. In Section 7, we introduce a tool named `stracematch`. This tool analyzes a log of system calls and verifies the correct usage of file descriptors in a real program. Finally, Section 8 concludes the paper.

2 Related work

Many specification-based runtime-monitoring frameworks have been proposed, including four approaches to parametric monitoring: an automaton-based approach [5, 7, 15, 16, 18, 20, 28, 34]; a regular expression- and grammar-based approach [1, 18, 24]; an approach based on temporal logic [6, 8, 9, 18–20, 24, 32, 33, 39, 48–50]; and a rule-based approach [4, 6, 35].

Qadeer and Tasiran [44] surveyed monitoring methods for multi-threaded programs. They identified several important classes of correctness criteria, such as race freedom, atomicity, serializability, linearizability and refinement. Their notion of refinement is different from the CSP context. In the CSP context, it is a relation between specification, while a refinement relation in Qadeer and Tasiran mean a relation between traces. Qadeer and Tasiran's paper [44] deals

with specific properties and construction of specialized monitors, not general-purpose specification languages like ours.

Among work on monitoring distributed systems, [10] and [23] concentrate global properties which can be described by temporal logic. polyLarva [17] is an extension of LARVA [15, 16] designed for distributed systems, by which a user can control the location of verifiers in the distributed system explicitly. LARVA specifies the properties of monitored systems by sets of automata which can communicate each other through channels. In this respect, it has a similar spirit to CSP_E, because both aim to synthesize a specification of a whole system from a specification of each component. However, LARVA is based on automata and does not support launching new automata dynamically.

Compared to these other frameworks, our approach creates a new avenue for specification languages. We employ a process calculus—namely, CSP—to describe a specification of a monitored program. The advantage of using process calculi is primarily that they are designed to describe a concurrent system, thereby providing a way to describe concurrency in a monitored system easily and directly. Another benefit is that process calculi, and in particular CSP, allow for automatic model checking [52, 53]. Model checkers can check the *refinement* relation between CSP specifications. A CSP specification is a refinement of another CSP specification when the former is a more detailed specification of the latter. Thus, by checking the refinement relation, we can guarantee that the monitor will check for desirable properties that otherwise might not be able to be monitored directly, especially when they are too general.

Theoretically, any model checker based on process calculi, such as CADP [25] can be used to offline runtime monitoring. In particular, CADP has a tool called SEQ.OPEN [24], which can convert traces to Labeled Transition System (LTS). A trace can be converted to an LTS and then synchronously composed to a specification. If the system in such a way exhibits a deadlock, this indicates that the trace violates the specification. The difference of our approach is that, CSP_E does not need a full model checker, and thus is much more lightweight. Also CSP_E is a DSL which is embedded in a general-purpose programming language, thus it is easy to extend. Finally, CSP_E can be used for online monitoring. To use model-checking approach above for online monitoring, a significant development effort seems necessary. "exhibitor" and "evaluator" of CADP can check whether traces confirm given specifications, if they are used together with SEQ.OPEN, but they use regular expressions or alternation-free μ-calculus for specification languages.

Implementation-wise, CSP_E is a shallow-embedded DSL in Scala [3]. In this respect, it closely resembles TraceContract [7]. However, the application program interface (API) of CSP_E is considerably different from that of TraceContract.

3 Introduction to CSP_E

In this section, we introduce CSP_E using a motivating example, and we compare it to QEA [5, 36, 46, 45]. We argue that CSP_E excels at describing properties that are composed of local properties (i.e., to a process) yet interact globally.

3.1 Motivating example

Our motivating example is a system of Unix-like processes and file descriptors. Each process and file descriptor have unique IDs. We assume that only Process 0 is running initially, and that Process 0 never exits. Each process can spawn child processes. Any process other than Process 0 can exit anytime, without waiting for its child processes to exit. Each process can open any file descriptor. If a process opens a file descriptor, it must close it before exiting. Opening the same file descriptor twice without closing it, or closing the same file descriptor twice without opening it again, is not allowed. If a file descriptor is opened by a process, the process can access it. Child processes can also access file descriptors that were opened by their parent before they were spawned. Such file descriptors must be closed by a child process before a child process exits.

3.2 CSP_E Syntax

CSP_E provides a DSL that is shallowly embedded in Scala. This DSL is also called CSP_E. Fig. 1 shown the above specification in CSP_E.

The notable classes that appear in the example are `Event` and `Process`.

Event. The monitored system creates a stream of *events*. CSP_E models events using the `Event` case class, which is a subclass of `AbsEvent`. Every event takes a symbol called an *alphabet* as its first argument, and it can have any number of trailing arguments of `Any` type.

Process. A *process* class and its subclasses in CSP_E are specifications of systems that are being monitored. They describe how a system produces sequences of events. A process is often termed a *monitor*, because a process that is constructed by the CSP_E accepts an event stream and judges whether it follows the specification that the process describes. A process is constructed using methods that are defined by the CSP_E library. In Fig. 1, `system` is the description of the entire system, while `sysproc` represents a process (and its subprocesses). `uniqProcess` guarantees that process IDs are all unique for all process. `P1 || Set('Spawn, 'Exit) || P2` means that `P1` and `P2` run concurrently and share events which have alphabets `'Spawn` and `'Exit`. `?? {case ... => ...}` is a pattern match on incoming events, `P1 ||| P2` is the interleaving of `P1` and `P2`, and `SKIP` is a process that does nothing and terminates normally. A process can be defined by recursion, using the standard Scala `def` syntax. For the full syntax, see Fig. 2 in BNF form.

Intuitively, each construct has the following meaning:

```
def sysproc(pid: Int, openFiles: Set[Int]): Process = ?? {
  case Event('Spawn, `pid`, child_pid: Int) =>
    sysproc(pid, openFiles) ||| sysproc(child_pid, openFiles)
  case Event('Open, `pid`, fd: Int) if !openFiles(fd) =>
    sysproc(pid, openFiles + fd)
  case Event('Access, `pid`, fd: Int) if openFiles(fd) =>
    sysproc(pid, openFiles)
  case Event('Close, `pid`, fd: Int) if openFiles(fd) =>
    sysproc(pid, openFiles - fd)
  case Event('Exit, `pid`) if pid != 0 && openFiles.isEmpty => SKIP
}
def uniqSysproc(pidSet : Set[Int]) : Process = ?? {
  case Event('Spawn, _, child_pid : Int) if ! pidSet(child_pid) =>
   uniqSysproc(pidSet + child_pid)
  case Event('Exit, pid : Int) if pidSet(pid) =>
   uniqSysproc(pidSet - pid)
}
def system = sysproc(0, Set.empty) || Set('Spawn, 'Exit) ||
             uniqSysproc(Set(0))
var monitors = new ProcessSet(List(system))
```

Fig. 1. Motivating example in CSP_E

- `SKIP` : A process that does nothing and terminates normally. For example, `e ->: SKIP` represents a process that accepts event `e` and terminates.
- `STOP` : A process that does nothing and never terminates. For example, `e ->: STOP` accepts event `e` but gets stuck.
- `Failure` : A process that has already failed. For example, `e ->: Failure` accepts event `e` and then fails immediately. Using a process `P` which does not consume `e`, the difference between `STOP` and `Failure` can be seen in the examples `e ->: STOP ||| P` and `e ->: Failure ||| P`. Both processes accept event `e`, but subsequently, the former runs provided that `P` accepts events, whereas the latter fails immediately after event `e`. Using `Failure` explicitly marks that a branch leads to an impossible state.
- `e ->: P` : A process that accepts event `e` and behaves like `P`. For example, `e1 ->: e2 ->: SKIP` accepts events `e1` and `e2` and then terminates.
- `?? (f : Event -> Process)` : A process that accepts event `e` and then behaves like `f(e)`. The function `f` typically uses pattern matching in order to select the behavior that matches that of `e`. If `e` does not match any pattern, `?? f` behaves like `Failure`.
- `??? (f : Event -> Process)` : Similar to `??`, but unlike `??`, if the incoming event `e` does not match any pattern, `??? f` waits for another event.
- `P1 <+> P2` : A process that behaves like either `P1` or `P2`, depending on which of them can accept the event that is input to `P1 <+> P2`. If both `P1` and `P2` can accept the event, both possibilities remain. For example,

$$
\begin{aligned}
P ::= & \texttt{SKIP} \mid \texttt{STOP} \mid \texttt{Failure} \mid \\
& e \texttt{ ->: } P \mid \\
& \texttt{??} f \mid \texttt{???} f \mid \\
& P \texttt{ <+> } P \mid \\
& P \texttt{ || } a \texttt{ || } P \mid \\
& P \texttt{ ||| } P \mid \\
& P \texttt{ \$ } P
\end{aligned}
$$

Fig. 2. Full syntax for CSP_E : Here, `SKIP`, `STOP`, `Failure` are constants, e is an event, f is a partial function which maps a event to a process, and a is a set of alphabets.

`e ->: e1 ->: SKIP <+> e ->: e2 ->: Failure` first accepts `e`. Then, if it accepts `e1`, it terminates normally; if it accepts `e2`, however, it fails immediately. If both processes cannot accept the event, it behaves like Failure.

- `P1 || a || P2` : Concurrent composition of `P1` and `P2`, using events whose alphabets are elements of `a` as synchronization events. For example, if `a` contains the alphabet of `e0` but `a` does not contain alphabets of `e1` and `e2`, then `e0 ->: e1 ->: SKIP || a || e0 ->: e2 ->: SKIP` accepts the sequences `e0, e1, e2` and `e0, e2, e1` but does not accept `e0, e0, e1, e2`.
- `P1 ||| P2` : Interleaving of `P1` and `P2`. `e1 ->: SKIP ||| e2 ->: SKIP` accepts the sequences `e1, e2` and `e2, e1`.
- `P1 $ P2` : A process that first behaves as `P1` and then behaves as `P2`. If `P1` fails, then `P1 $ P2` fails. For example, `e1 ->: SKIP $ e2 ->: SKIP` accepts `e1` and `e2` and then terminates normally.

Then, we generate a set of monitors with `new ProcessSet(...)` constructs. Because CSP_E can track multiple possible states in the monitored system, we use the set of monitors to monitor the system. Our system is similar to the one presented in [37], but with unique differences, such as the inclusion of `Failure`. Another difference is that our system has no internal choice, no τ, and no hiding. Theoretically, we can implement such constructs, but they easily lead to the state explosion of the monitors in which they are used. We designed the syntax of CSP_E in such a way that it closely resembles the original CSP syntax. However, there are syntactical limitations, owing to the fact that CSP_E is embedded [3] in Scala. In Scala, the associativity and precedence of operators are both determined by the established syntax of operators, and thus cannot be changed. For example, to make the operator $\rightarrow$ right-associative, we need to add `:` to `->`. Moreover, some symbols in Scala, such as `[]`, are reserved for certain kinds of operations. Thus, we use `<+>` rather than `[]` to indicate the choice operator. In the traditional CSP, the human-readable representation differs from the machine-readable representation. For CSP_E, we selected the machine-readable code as a common representation for both types for simplicity. Because the

human-readable representation of CSP_E and its embedding to Scala are the same, both are termed CSP_E.

3.3 Comparison with QEA

Because our specification contains a specification on file descriptors and a specification on processes, we split these specifications into two QEA automata: `fdMonitor` and `processMonitor`. Using the textual representation used in [36], each can be defined as shown in Fig. 3.

```
qea {
  Forall(fd)
  accept skip(init) {
    spawn(parent, child) if [ parent in PS ]
      do [ PS.add(child) ] -> init
    open(pid, fd) if [ pid in PS ] -> failure
    open(pid, fd) if [ not pid in PS ]
      do [ PS.add(pid) ]-> init
    access(pid, fd) if [ not pid in PS ] -> failure
    close(pid, fd) if [ pid in PS ]
      do [ PS.remove(pid) ] -> init
    close(pid, fd) if [not pid in PS ] -> failure
    exit(pid) if [ pid in PS ] -> failure
  }
}

qea {
  accept next(init) {
    spawn(parent, child)
      if [ (parent in PS || parent = 0) && not child in PS && child = 0]
      do [ PS.add(child) ] -> init
    exit(pid) if [ pid in PS ]
      do [ PS.remove(pid) ] -> init
    open(pid, fd) -> init
    access(pid, fd) -> init
    close(pid, fd) -> init
    }
}
```

Fig. 3. QEA monitor for file descriptors and processes, respectively

Theoretically, CSP_E, and QEA can represent any Turing computable property, because we allow CSP_E and QEA to have rich data-structures such as `Set`; thus, we can encode a universal Turing machine using these data structures as tapes. However, such theoretical representation is of no practical interest, because such a monitor does not have much difference to a hand-coded monitor.

A major characteristic of CSP_E is that it can represent the specification in a bottom-up manner. The entire monitor is built from `sysproc(pid, openFiles)` monitors, which run in an interleaving fashion, and `uniqProcess`, which guarantees that each process has a unique ID. Moreover, `sysproc(pid, openFiles)` can be further decomposed into concurrent monitors. Thus, it can describe the specification in the term near its implementation yet abstracted from unnecessary details. A major characteristic of QEA is that, insofar as it is based on state machines, it requires specifying the states and transitions of the whole system in order to describe the specification.

4 Formal Semantics

In this section, we define the formal semantics of CSP_E. Because it is outside the scope of this paper to define a language that contains all of the Scala constructs, we consider a language that consists of all the constructs explicitly described in Section 3.2, along with purely functional Scala expressions.

In the context of a runtime-monitoring framework, the term *formal semantics* refers to the formal definition of actions of a monitor when it is given a finite sequence of events (trace). For each trace, there are three possibilities:

1. The trace is accepted as a successful and complete execution of the monitored program.
2. The trace is successful thus far, but not yet complete.
3. The trace is rejected as a failure.

Every possible trace falls into one of these three categories. To distinguish these three cases, it is enough to give a set of traces which satisfy either cases 1 or 2. We call such a set a *trace semantics* of a monitor. In the given trace semantics T, we interpret each trace as follows:

1. A trace $t \in T$ that ends with ✓ is interpreted as a successful and complete execution.
2. A trace that is contained in T and does not end with ✓ is interpreted as an incomplete trace.
3. A trace that is not contained in T is interpreted as a failure.

The semantic space $\mathcal{T}$ of all possible trace semantics consists of the set of traces T that satisfies the following:

1. If $t \in T$ and t' is a prefix of t, then $t' \in T$, and
2. for all $t \in T$, the ✓ appears only at the end of t.

This definition allows for the $\emptyset$ to be included as a member of $\mathcal{T}$, unlike the usual definition of CSP trace semantics [37, 47]. Let T^c denote all completed traces in T, T^i denote all incomplete traces in T, and T^f denote all traces that are not contained in T. We allow for the possibility that a trace $t \in T$ is interpreted as successful thus far, despite the failure of all of its successors—for example, when

$\langle\rangle \in$ `STOP`. From here, we abuse notations and use `STOP`, `SKIP` and `Failure` to denote their semantics. The `STOP` specification is required to be used for concurrent composition. If `STOP` is coupled with other processes concurrently, `STOP` will get stuck, whereas other processes will continue to run. If `Failure` is coupled with other processes concurrently, on the other hand, `Failure` causes an immediate failure in the entire system.

Because we allow for a recursive definition of monitors, fixed points of reasonable sets of operators are needed in the semantic space $\mathcal{T}$. We follow the usual approach by introducing a structure known as a complete partial order (CPO) into the semantic space $\mathcal{T}$. We define the order relation $T_1 \sqsubseteq T_2$ to hold for elements T_1, T_2 in $\mathcal{T}$, if $T_1 \subseteq T_2$. CPO is a partial order of which directed subsets have supremums. A directed subset is a subset of a partial order such that it is non-empty and every pair in it has an upper bound in it. A complete lattice is a partially ordered set of which all subsets have supremums and infimums. If all subsets have supremums, they also have infimums [47].

Theorem 1. *$\mathcal{T}$ is a complete lattice.*

Proof. For a set $(T_i)_{i\in X}$ of elements in $\mathcal{T}$, its least upper bound $T = \bigsqcup_{i\in X} T_i$ is defined as $T = \bigcup_{i\in X} T_i$ (i.e., the union of $(T_i)_{i\in X}$ as sets). It is routine to check whether $T \in \mathcal{T}$.

It is well known that all complete lattices are CPO. We define $T_1 \sqcup T_2$ as $\bigsqcup_{i=1,2} T_i$. Here, $\mathcal{T}$ has the minimal element `Failure`, and `Failure` does not contain any trace. Moreover, `STOP` consists solely of $\langle\rangle$. In the usual trace semantics, `STOP` is the smallest element of the semantic space, but it is clear that `Failure` $\sqsubset$ `STOP`. We also define `SKIP` as `SKIP` $= \{\langle\rangle, \langle\checkmark\rangle\}$. Next, we define the operators $\rightarrow, \frown$ on $\mathcal{T}$ in order to define the semantic interpretations of `->:` and `$`. We use the operation $e^\frown s$ and $s \frown t$ on an event e and traces s, t in order to define $\rightarrow$ and $\frown$. Here, $e^\frown s$ of the concatenation of the event e to s, and $s \frown t$ is the concatenation of s and t, but if s ends with a $\checkmark$, then this $\checkmark$is removed. Let e denote an event other than $\checkmark$, and let $T, T_1, T_2 \in \mathcal{T}$.

$$e \rightarrow T = \{\langle\rangle\} \cup \{e^\frown t \mid t \in T\}$$
$$T_1 \frown T_2 = T_1^i \cup \{s \frown t \mid s \in T_1^c, t \in T_2\}$$

Theorem 2. *If e is an event and $T, T_1, T_2 \in \mathcal{T}$, then $e \rightarrow T, T_1 \frown T_2 \in \mathcal{T}$.*

Next, we define the concurrent composition $T_1 ||_A T_2$ ($\checkmark \notin A$) by co-induction on T_1 and T_2. $T \in \mathcal{T}$ is either empty ($T =$ `Failure`) or it can be decomposed as follows:

$$C \sqcup \bigsqcup_{e \in \mathrm{hd}(T)} e \rightarrow T^e \tag{1}$$

where C is `SKIP` if T can successfully terminate immediately or `STOP` otherwise. Here, $\mathrm{hd}(T)$ is defined as $\{e \mid \exists t, e^\frown t \in T\}$. Intuitively, $T_1 ||_A T_2$ is defined as follows: first, we assume that T_1 and T_2 allow only one event, e_1 for T_1 and e_2

for T_2 at the first step, and after these events their behaviors are described by $T_1^{e_1}$ and $T_2^{e_2}$ respectively. If both events are not in A, we assume that $T_1||_A T_2$ behaves in an interleaved way, that is, $T_1||_A T_2 = e_1 \to (T_1^{e_1}||_A T_2) \sqcup e_2 \to (T_1||_A T_2^{e_2})$. If one of them, say, e_2 fall into A but not e_1, then first e_1 occurs and the occurrence of e_2 is delayed $e_1 \to (T_1^{e_1}||_A e_2 \to T_2^{e_2})$. If both fall into A, then they must be equal otherwise $T_1||_A T_2 = \texttt{STOP}$. If $e = e_1 = e_2$, $T_1||_A T_2 = e \to (T_1^e||_A T_2^e)$. If there are multiple possibilities for events which can occur at the first step, we can write $T_1 = \bigsqcup_{e \in \mathrm{hd}(T_1)} e \to T_1^e$. We define $T_1||T_2$ by distributivity: $T_1||_A T_2 = \bigsqcup_{e \in \mathrm{hd}(T_1)} e \to T_1^e||_A T_2$. Considering the possibility of immediate termination, we reach the following conditions. Let $a, a' \in A$ and $b, b' \notin A \cup \{\checkmark\}$.

1. $\texttt{Failure}||_A T = \texttt{Failure}$
2. $\texttt{SKIP}||_A T = T$
3. $\texttt{STOP}||_A T = T \backslash T^c$
4. $a \to T_1^a||_A a \to T_2^a = a \to (T_1^a||_A T_2^a)$
5. $a \to T_1^a||_A a' \to T_2^a = \texttt{STOP}$ if $a \neq a'$
6. $b \to T_1^b||_A a \to T_2^a = b \to (T_1^b||_A a \to T_2^a)$
7. $b \to T_1^b||_A b' \to T_2^{b'} = b \to (T_1^b||_A b' \to T_2^{b'}) \sqcup b' \to (b \to T_1^b||_A T_2^{b'})$
8. $(C \sqcup \bigsqcup_{e \in \mathrm{hd}(T_1)} e \to T_e)||_A T_2 = (C||_A T_2) \sqcup \bigsqcup_{e \in \mathrm{hd}(T_1)} (e \to T_e||_A T_2)$
9. All symmetric cases of above rules.

Theorem 3. *If $T_1, T_2 \in \mathcal{T}$, $T_1||_A T_2$ exists and is an element of $\mathcal{T}$.*

Theorem 4. *The operators $\sqcup, \bigsqcup, \to, ||_A$, and $\frown$ are all monotonic.*

Thus, Tarski's fixed-point theorem can be applied, and the use of recursion is justified [47].

We interpret `Failure`, `STOP`, and `SKIP` in the CSP_E notation as `Failure`, `STOP`, and `SKIP` in $\mathcal{T}$, respectively, and `->:`, `<+>`, `?? f`, `|| a ||`, and `$` as $\to$, $\sqcup$, $\bigsqcup_{e:\texttt{Event}} e \to \texttt{f}(e)$, $||_{A(a)}$, and $\frown$, respectively. Here, $A(a)$ is the set of events that have elements of `a` as alphabets. Moreover, `???` and `|||` can be defined by using the operators above and a recursive definition.

5 Implementation

In this section, we discuss an implementation of CSP_E, which is available at [54]. CSP_E was implemented as an internal shallow-embedded DSL in the Scala programming language. Artho et al. discussed in depth the use of a DSL in the context of verification [3]. The proposed CSP_E is a combinator library that can be used to define an object of the `Process` class, which represents a monitor. An object of the `Process` class includes a method called `accept`, which accepts an event object and returns new monitors. To monitor an event sequence, `ProcessSet` is first instantiated by a list of monitors that are defined by CSP_E. `ProcessSet` also includes a method named `accept`, which accepts an event and returns a new `ProcessSet`. `ProcessSet` represents multiple possible system states that are being monitored. Each monitor contained in `ProcessSet` represents one possible

system state. `ProcessSet` includes a method named `isFailure`, which returns a Boolean value. If it returns true, there is no possible state that can be interpreted as a correct system state, thus signifying the occurrence of an error. `ProcessSet` includes a method named `canTerminate`, which also returns a Boolean value. If it returns true, the system can exit at that point. If it returns false and the system exits, the exit is an error.

As stated above, the `Process` object returns new `Process` objects when it accepts a new event, rather than changing its internal state. This design simplifies the implementation of non-determinism and concurrency. However, it also requires immutable data structures for the internal state of a `Process` object. A data structure for holding sets of processes is crucial in terms of performance, especially when there are many processes running concurrently. The current implementation uses the standard Scala List, which allows for duplicate entries. A set in the standard library and multisets [51] have more efficient representations, but we found that using them degrades the performance. Removing duplicate entries appears to be unnecessary, because the current implementation generates many closures. Closures in Scala are compared by their physical equality, thus all closures which are generated at different times are distinct. Thus, the considerable amount of time needed to compute hashes and compare objects is superfluous. Furthermore, we attempted to enhance the performance and reduce memory usage by caching the results of `accept`. Alas, this slowed the library, contrary to our expectations. In that experiment, we used ScalaCache [11] together with Google Guava [27].

6 Benchmark

We compared the performance of our implementation of CSP_E with an implementation of QEA [14], using a randomly generated event sequence that correctly simulates our motivating example. The benchmark generated an event sequence and fed this into the CSP_E and QEA monitors inside the same program, without using an external log file. The program generated a sequence of `Event` objects using a `Stream` and put its first 300,000 elements into a `List` to avoid recording the time required to generate the sequence in the benchmark. Then, the benchmark program first fed these elements into the CSP_E monitor, recording the real time from the first event to each 1,000 events. The benchmark program also did the same to the QEA monitor, and finally formatted the result into the CSV format. The time required for the JVM startup and the initialization of the monitors is not included. The benchmark program is included in the `test` directory of the CSP_E distribution [54].

The benchmark uses Java `1.8.0_77-b03` and Scala 2.11.8 on Mac OS X El Capitan, which we executed on a Mac Pro (late 2013 model), with a 3.7 GHz Quad-Core Intel Xeon E5 and 64 GB of 1866 MHz DDR3 ECC.

Fig. 4 shows the results of this experiment. The time required to process n events is roughly quadratic to n, because the number of processes in the monitored system increases linearly to n and for each event, checking that every

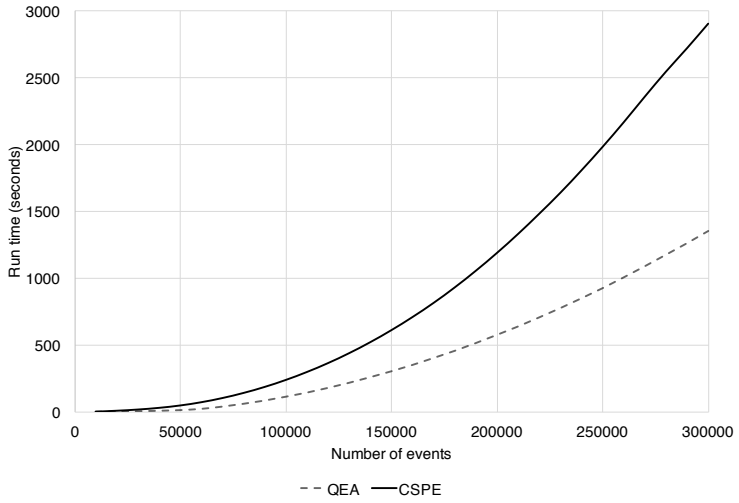

Fig. 4. Benchmark with the motivating example

process follows the specification requires time linearly to n. We can observe that QEA is slightly more than twice as fast as CSP_E. Nevertheless, because the current implementation of CSP_E has not yet been optimized, this shows that CSP_E is a feasible approach to monitoring concurrent systems.

7 Application: stracematch

`stracematch` is contained in the example directory of the CSP_E distribution [54]. CSP_E is used to implement a complete model of open and closed system calls for file descriptors, the `fork` system call, and the `execve` system call. It accepts the `dtruss -f` output on OS X from the standard input, and then verifies that the program correctly handles file descriptors. `dtruss` is based on `DTrace` [12]. `DTrace` can instrument kernel events using `D`-script (not related to the `D` programming language), and do an arbitrary complex task at each kernel event.

`dtruss` is a wrapper of `DTrace` and traces system calls and produces a log, similar to `strace` in Linux. However, there are limitations: 1) `dtruss` cannot obtain file descriptors which are created by `socketpair` and `pipe`, 2) `dtruss` cannot catch the invocation of `execve` system call, which causes errors, 3) `dtruss` outputs the trace to `stderr`, while it also outputs error messages to `stderr`, therefore, if an error happens, the error message has to be removed from the log manually. In the future work, we may consider using directly `DTrace` to circumvent these limitations.

Owing to these limitations of `dtruss`, `stracematch` verifies that (a) a process does not close the same file descriptor twice, and (b) if `stracematch` detects

the opening of a file descriptor, it is closed in the same thread. We applied `strace match` to several programs, as indicated in Table 1, where the "log size" represents the number of lines in the log, and "time" denotes the amount of time in seconds that `strace match` required to analyze the log, which includes the time to start the JVM.

Table 1. Benchmark of strace match

Program	Log size	Result	Time [s]
`ls`	156	Passed	0.370
`wget` (short)	267	Fd not closed	0.377
`wget` (long)	28,901	Fd not closed	1.145
`Emacs`	2,678	Fd not closed	0.750
`Chrome`	166,090	Stopped at 2,935	0.810
`Ruby ehttpd`	11,034	Closed fd twice at 1,168	0.648
`Sinatra`	3,191	Closed fd twice at 1,170	0.673
`bash`	1,218	Closed fd twice at 946	0.575

ls [2] is a Unix command which lists the contents of a directory. "ls" shows the result on a trace which was obtained by invoking `ls -l` in a directory. wget [31] is a command line tool to download the contents of a Web page or a Web site. "wget (short)" shows the result on a trace which was obtained by downloading a single web page. "wget (long)" shows the result on a trace which is obtained by downloading an entire web site recursively. Emacs [30] is a popular text editor. "Emacs" shows the result on a trace which was obtained by invoking and terminating Emacs on the terminal. Chrome [26] is a popular Web browser. "Chrome" shows the result on a trace which was obtained by starting and exiting Chrome. Ruby ehttpd is a web server which is included in the standard Ruby [40] programming language distribution. Sinatra [42] is a simple web framework. "Ruby ehttpd" and "Sinatra" shows the results on traces which were obtained by running these web servers on a simple static web site and downloading the entire web site by wget. bash is a popular Unix command shell. "bash" [29] shows the result on a trace which was obtained by running an artificial shell script which was extracted from `sbt`, a build system which is used mainly for Scala programming language. The shell script which was used, is available upon request.

Our tool can process any logs within less than 1.2 seconds. With the exception of the case of "ls", `strace match` ended with errors. When `strace match` encounters an error, it stops and prints the line number and the contents of the line. In Table 1, the location of each termination is indicated by the line number at which `strace match` stopped. One group of errors involved some file descriptors that were still open at the end of the execution. This may be an indication of a leak of that file descriptor. Another group of errors involved closing the same file descriptor twice without re-opening it in between. This can cause a rare bug in which another thread or a signal handler opens a file descriptor after

that file descriptor is closed for the first time and then the same file descriptor is closed again; thus, that other thread or that signal handler cannot use this file descriptor after that point. In the "Chrome" case, `stracematch` stopped because invocations of `fork` appeared in the log, but only for child processes; and there were no record of the invocation of `fork` for the parent processes. In such a case, `stracematch` will become confused and stop working.

We contacted the developers of `wget` and `bash`, but the `bash` developer answered that the error is harmless. The `wget` developers did not respond.

8 Conclusion

In this paper, we presented CSP_E, an event-monitoring framework for concurrent systems inspired by Hoare's CSP. Monitoring concurrent systems is important, because concurrent systems are now ubiquitous, and unit tests are incapable of detecting concurrency bugs. Unlike most of other monitoring frameworks, CSP_E can describe a concurrent system in a bottom-up fashion, by composing specifications of system components. This is often a natural way of specifying concurrent systems.

CSP_E is implemented as an internal DSL in Scala. Consequently, it can be easily adapted to the needs of specific applications. Although the current implementation has not yet been optimized, it incurs acceptable overhead, as we demonstrated by comparing it to a QEA implementation to monitor a simulated event sequence of a highly concurrent system. We presented the formal semantics for CSP_E in Section 4. Thus, CSP_E is amenable to theoretical analysis. The semantics are closely similar to the standard trace semantics of CSP. To interpret the `Failure` construct, however, we allow the $\emptyset$ as a valid element in the semantic space. We showed that even in such a setting, the semantic space forms a CPO, thus allowing for fixed-point construction.

There are several directions for future research. Currently, a CSP_E monitor can only determine whether a given event trace is correct, outputting a Boolean value. This makes it difficult to find the cause of an error. In addition, there is no mechanism for error recovery. Rather, a monitor simply terminates when it first encounters an error. We plan to add functionality that records the event sequence to the error, providing error recovery based on diagnostics. CSP_E does not currently offer an easy way to share states among monitors running concurrently. Global variables in Scala cannot be used for this purpose, because they are also shared between monitors that represent the different states of the system. We shall consider a mechanism and a language feature that can record the global states among monitors.

Acknowledgments: We are grateful to Giles Reger for helping to develop the QEA models, and to Eijiro Sumii for helping us to develop the proof of Theorem 3. Yoshinao Isobe influenced the early design of the language. This work was supported by JSPS KAKENHI Grant Number JP26280019. We would like to thank Editage (www.editage.jp) for English-language editing.

References

1. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O.V.R., De Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: Johnson, R., Baniassad, E. Gabriel, R. P., Noble, J., Marick B. (eds.) OOPSLA'05. pp. 345–364, ACM New York (2005)
2. Apple: ls, version 7.2.0.0.1.1447826929
3. Artho, C., Havelund, K., Kumar, R., Yamagata, Y.: Domain-specific languages with Scala. In: Butler, M., Conchon, S., Zaïdi, F, (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 1–16. Springer International Publishing (2015)
4. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for run-time monitoring: from EAGLE to RULER. Journal of Logic and Computation 20(3), 675–706. Oxford University Press (2010)
5. 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. LNCS, vol. 7436, pp. 68–84. Springer Berlin Heidelberg (2012)
6. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) Verification, Model Checking, and Abstract Interpretation, LNCS, vol. 2937, pp. 44–57. Springer Berlin Heidelberg (2004)
7. Barringer, H., Havelund, K.: TraceContract: A Scala DSL for trace analysis. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, pp. 57–72. Springer Berlin Heidelberg (2011)
8. Basin, D., Klaedtke, F., Müller, S.: Policy monitoring in first-order temporal logic. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 1–18. Springer Berlin Heidelberg (2010)
9. Bauer, A., Küster, J.C., Vegliach, G.: From propositional to first-order monitoring. In: Legay, A., Bensalem, S. (eds.) Runtime Verification. LNCS, vol. 8174, pp. 59–75. Springer Berlin Heidelberg (2013)
10. Bauer, A., Falcone, Y. (2012).: Decentralised LTL monitoring. In: Giannakopoulou, D., Méry D. (eds.) FM 2012. NCS, vol. 7436, pp. 85-100. Springer Berlin Heidelberg (2012)
11. Birchall, C.: ScalaCache, https://github.com/cb372/scalacache
12. Cantrill, B., Shapiro, M., Leventhal, A.: Dynamic Instrumentation of Production Systems. In: USENIX 2004, pp. 15–22. USENIX (2004)
13. Chen, F., Roşu, G.: Parametric trace slicing and monitoring. In: Kowalewski, S., Philippou, A. (eds.) TACAS'09. LNCS, vol. 5505, pp. 246–261. Springer Berlin Heidelberg (2009)
14. Cuenca, H.: QEA, https://github.com/selig/qea
15. Colombo, C., Pace, G. J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: Darren C., Alessandro F. (eds.) FMICS 2008. LNCS, vol 5596, pp. 135–149. Springer Berlin Heidelberg (2009
16. Colombo, C., Pace, G. J., Schneider, G.: LARVA - Safer monitoring of real-time Java programs (tool paper). In: Hung, D. V., Krishnan, P. (eds.) SEFM 2009. IEEE Computer Society (2009)
17. Colombo, C., Francalanza, A., Mizzi, R., Pace, G. J.: PolyLarva: Runtime verification with configurable resource-aware monitoring boundaries. In: Eleftherakis, G., Hinchey, M., Holcombe M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 218-232. Springer Berlin Heidelberg (2012)

18. D'Amorim, M., Havelund, K.: Event-based runtime verification of Java programs. ACM SIGSOFT Software Engineering Notes 30(4), ACM New York (2005)
19. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. In: Ábrahám, E., Havelund, K. (eds.) TACAS'14. LNCS, vol. 8413, pp. 341–356. Springer Berlin Heidelberg (2014)
20. Drusinsky, D.: Modeling and verification using UML statecharts: a working guide to reactive system design, runtime monitoring and execution-based model checking. Newnes (2011)
21. Falcone, Y., Fernandez, J.C., Mounier, L.: Runtime verification of safety-progress properties. In: Bensalem, S., Peled, D. (eds.) Runtime Verification. LNCS, vol. 5779, pp. 40–59. Springer Berlin Heidelberg (2009)
22. Runtime Verification 2014: first international competition on Runtime Verification, http://rv2014.imag.fr/monitoring-competition/results.html
23. Francalanza, A., Seychell, A.: Synthesising correct concurrent runtime monitors. Formal Methods in System Design, 46(3), 226-261. Springer US (2014)
24. Garavel, H.,Mateescu, R.: SEQ . OPEN : a tool for efficient trace-based verification. In: Graf, S. and Mounier, L. (eds.) International SPIN Workshop on Model Checking of Software. LNCS, vol 2989, pp. 151–157. Springer Berlin Heidelberg (2004)
25. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: A toolbox for the construction and analysis of distributed processes. International Journal on Software Tools for Technology Transfer, 15(2), pp. 89-107. Springer (2013)
26. Google Inc.: Chrome, version 47.0.2526.111 (64-bit)
27. Google Inc.: Guava, https://github.com/google/guava
28. Goubault-Larrecq, J., Olivain, J.: A smell of orchids. In: Leucker, M. (ed.) Runtime Verification, LNCS, vol. 5289, pp. 1–20. Springer Berlin Heidelberg (2008)
29. GNU project: bash, version 4.3.42(1)-release (x86_64-apple-darwin14.5.0), https://www.gnu.org/software/bash/
30. GNU project: Emacs, version 24.5.1, https://www.gnu.org/software/emacs/
31. GNU project: wget, version 1.17.21-df7cb-dirty built on darwin14.5.0., https://www.gnu.org/software/wget/
32. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Transactions on Services Computing 5(2), pp. 192–206. IEEE (2012)
33. Havelund, K., Roşu, G.: Monitoring programs using rewriting. In: Feather, M., Goedicke, M. (eds.) ASE 2001. pp. 135–143, IEEE CS Press, (Nov 2001)
34. Havelund, K.: Runtime verification of C programs. In: Suzuki, K., Higashino, T., Ulrich, A., Hasegawa, T. (eds.) TESTCOM, LNCS, vol. 5047, pp. 7–22. Springer Berlin Heidelberg (2008)
35. Havelund, K.: Rule-based runtime verification revisited. International Journal on Software Tools for Technology Transfer 17(2), pp. 143–170. Springer (2014)
36. Havelund, K., Reger, G.: Specification of parametric monitors: Quantified event automata versus rule system. In: Formal Modeling and Verification of Cyber-Physical Systems, pp. 151–189. Springer Fachmedien Wiesbaden (2015)
37. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall International, London (Aug 1985)
38. Kassem, A., Falcone, Y., Lafourcade, P.: Monitoring electronic exams. In: Bartocci, E., Majumdar, R. (eds.) Runtime Verification 2015. LNCS 9333, pp. 118–135. Springer International Publishing (2015)

39. Lee, I., Kannan, S., Kim, M., Sokolsky, O., Viswanathan, M.: Runtime assurance based on formal specifications. In: Arabnia, Hamid R. (eds.) PDPTA 1999, pp. 279–287. CSREA Press (1999)
40. Matsumoto, Y.: Ruby, version 2.2.2p95 (2015-04-13 revision 50295) [x86_64-darwin14]
41. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. International Journal on Software Tools for Technology Transfer 14(3), pp. 249–289. Springer (2012)
42. Mizerany, B.: Sinatra, version 1.4.7, http://www.sinatrarb.com/
43. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima, 2016
44. Qadeer, S., Tasiran, S.: Runtime verification of concurrency-specific correctness criteria. International Journal on Software Tools for Technology Transfer, 14(3), 291-305. Springer (2012)
45. Reger, G.: Automata based monitoring and mining of execution traces. Ph.D. thesis, University of Manchester (2014)
46. Reger, G., Cruz, H.C., Rydeheard, D.: MarQ: Monitoring at Runtime with QEA, In: Baier, C., Tinelli, C. (eds.) TACAS 2015, LNCS, vol. 9035, pp. 596–610. Springer Berlin Heidelberg (2015)
47. Roscoe, A.W., Hoare, C.A.R., Bird, R.: The Theory and Practice of Concurrency. Prentice Hall PTR (1997)
48. Stolz, V.: Temporal assertions with parametrized propositions. Journal of Logic and Computation 20(3), pp. 743–757, Oxford University Press (2008)
49. Stolz, V., Bodden, E.: Temporal assertions using AspectJ. Electronic Notes in Theoretical Computer Science 144, pp. 109–124, Elsevier (2006)
50. Stolz, V., Huch, F.: Runtime verification of concurrent Haskell programs. Electronic Notes in Theoretical Computer Science 113, pp. 201–216, Elsevier (2005)
51. Stucki, N.: multisets, https://github.com/nicolasstucki/multisets
52. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards flexible verification under fairness, In: Bouajjani, A., Maler, O. (eds.) CAV 2009, LNCS, vol. 5643, pp. 709–714. Springer Berlin Heiderberg (2009)
53. University of Oxford: FDR3, https://www.cs.ox.ac.uk/projects/fdr/
54. Yamagata, Y.: `CSP_E`: Log analyzing tool for concurrent systems, https://github.com/yoriyuki/cspe