Strategy-Based Contract Monitors, Monitoring Meta-strategies, and Runtime Instrumentation

Cameron Swords
Indiana University
cswords@indiana.edu

Abstract

Contract systems have come to play a vital role in many aspects of software engineering, which has given rise to a myriad of contract monitoring strategies, ranging from Eiffel's classic pre- and post-condition contract enforcement to lazy, parallel, and even "best-effort" contracts. More recent work introduces a strategy-parameterized, unified framework for these approaches. We present a Clojure-based implementation of this multi-strategy approach, extended with *meta-strategies*, strategy-parameterized strategies that perform additional operations as part of enforcement. We also demonstrate how meta-strategies may be used to in a strategy-parameterized contract system to optimize contract enforcement, reason about global program behavior, and even perform runtime instrumentation.

CCS Concepts •**Theory of computation → Program reasoning;** •**Software and its engineering → Language features;**

Keywords runtime verification, spot-checking, lazy monitoring, behavioral specification, code tracing

ACM Reference format:
Cameron Swords. 2017. Strategy-Based Contract Monitors, Monitoring Meta-strategies, and Runtime Instrumentation. In *Proceedings of Dynamic Languages Symposium, Vancouver, BC, Canada, October 24, 2017 (DLS'17),* 12 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Software contracts were originally presented by Meyer [23] in the Eiffel language, and have since become an invaluable utility in dynamically-typed languages. The basic premise is that the user writes some collection of predicates that, taken together, dictate which values are allowed to flow through a program. Contract systems ensure the program adheres to this behavior by decorating existing programs with these predicates and checking them at the appropriate points.

However, the "appropriate point" to verify each predicate has a number of interpretations that impact the program in different ways, from over-evaluation to under-verification, and the verification method is often fixed on a system-wide scale [5, 7]. Swords et al. [28] present an alternative framework for contract verification wherein the contract assertion form is parameterized by a *monitoring strategy* that dictates *how* and *when* to enforce each *individual* contract, recovering a bounty of previously-published monitoring behaviors by viewing contracts as separate evaluators that interact with the user program.

This strategy-parameterized monitoring system, however, has two major drawbacks: first, the monitoring mechanism is a fixed term-rewriting form defined at the language level, so users may not define their own verification strategies, and, second, the strategies they present are all first-order. This work aims to address both of these concerns by presenting the MCON library, an extensible implementation of the formal system presented by Swords et al. [28] with *meta-strategies*, or strategy-parameterized strategies. As we will demonstrate, these meta-strategies have immense utility, including verification optimization, global verification reasoning, and even allowing programmers to use the contract system for general runtime instrumentation.

Our contributions include:

- introducing monitoring meta-strategies as a principled mechanism for handling contracts that encompass program state and protocol enforcement;
- the MCON library, an implementation of multi-strategy and meta-strategy monitoring in Clojure[1];
- a number of in-depth examples demonstrating the usage of verification meta-strategies, including recovering previous, ad-hoc verification mechanisms from the literature;
- and a discussion of how this system may be used to implement general runtime instrumentation, including function profiling and tracing.

Our paper proceeds as: we take a tour of the MCON system, demonstrating how multiple strategies and meta-strategies provide extensive verification support and open the door for general runtime instrumentation[2] (§2); describe our implementation (§3); explore multi-strategy monitoring, additional runtime monitoring, and instrumentation mechanisms in MCON (§4); and discuss related work and conclude (§5-6).

DLS'17, Vancouver, BC, Canada
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

[1] http://github.com/cgswords/mcon/
[2] Our presentation elides blame for brevity; we provide a second implementation with explicit blame parameters and *indy*-style blame as blame.clj in the repository [11].

2 The Good, The Extensible, and the Meta

We start with some examples to give a feel for MCON and demonstrate its use, including strategy-parameterized contract monitoring, monitoring meta-strategies, and how these mechanisms support general runtime instrumentation.

2.1 Strategy-Parameterized Verification (The Good)

We start by defining some contracts. To do this, we first introduce the predicate contract combinator predc, which takes a *predicate* (a function that return a Boolean value) and returns a contract that ensures that predicate holds. For example, we can define a predicate contract to ensure that a term is a natural number:

```
1 (def natc (predc (fn [x] (≥ x 0))))
```

To monitor this predicate contract, we invoke **mon** with the contract and a monitoring strategy (in this case, **eager**, which immediately enforces the contract):

```
1 > (+ 5 (mon eager natc 5))
2 ⇒10
3
4 > (+ 5 (mon eager natc -1))
5 ⇒Contract Violation: `-1` violated `natc`.
```

As with previous contract systems [14, 23], MCON raises an error when it detects a violation and, otherwise, returns the monitored term (potentially wrapped in further checks, as we will see later); the error at line 5 indicates that the value -1 violated the contract natc. The **eager** strategy follows the enforcement method described by Meyer [23], wherein the monitor interrupts the user evaluator, suspending the user program while the monitor enforces the contract and resuming with the result (or raising an error) when the monitor has completely verified the contract.

The MCON library provides a number of additional strategies, including **futur**, **semi**, and **conc**, which evaluate the contract concurrently in a computational future, layer-by-layer via delayed checks, and concurrently without later synchronization, respectively[3].

```
1 > (mon futur natc 5)
2 ⇒<future ...>
3
4 > @(mon futur natc 5)
5 ⇒5
6
7 > (mon semi natc -1)
8 ⇒<delay ...>
9
10 > @(mon semi natc -1)
11 ⇒Contract Violation: `-1` violated `natc`.
12
13 > (mon conc natc -1)
14 ⇒-1
15    Contract Violation: `-1` violated `natc`.
```

[3]We use Clojure's **@** to extract the result from both computational futures and delayed references.

Each strategy produces a different contract monitoring mechanism[4], allowing programmers to decide *how* and *when* to enforce contracts on a per-contract level [28]:

- The **futur** strategy creates a computational future [16], concurrently enforcing the contract [10, 28]. When the user retrieves the future's result, they either get back the contracted value or an error.
- The **semi** strategy creates a delayed computation that, when forced, suspends the user evaluator and performs contract enforcement, similar to semi-eager monitoring [5, 18].
- The **conc** strategy, described by Swords et al. [28], performs concurrent verification without later synchronization: it spawns a new thread, computes the contract, and either terminates with a value or error. In Clojure, this allows the original program to continue while reporting results "on the side," (while the user process continues with the original value).

In the context of structural contracts, first-class strategies provide us precise control over how and when *entire structures* may be explored by the contract system. For example, we may define a strategy-parameterized contract for pairs[5], ensuring each element is a natural number:

```
1 (defn natpairc [strat] (pairc strat natc natc))
```

The pairc combinator takes three arguments: the *strategy* to use when enforcing its subcontracts, the contract to check on the first subcomponent of the pair, and the contract to check on the second subcomponent of the pair[6]. The resultant contract, natpairc, takes its enforcement strategy as input. This is a key advantage of MCON: strategies are first-class values, and we may pass them around and parameterize contracts by *how* they should enforce their subcontracts.

To check this contract, we use **mon** and natpairc with a subcontract strategy:

```
1 > (first (mon eager (natpairc eager) (list 5 -1)))
2 ⇒Contract Violation: `-1` violated `natc`
3
4 > @(first (mon eager (natpairc semi) (list 5 -1)))
5 ⇒5
6
7 > (first @(mon futur (natpairc eager) (list 5 -1)))
8 ⇒Contract Violation: `-1` violated `natc`
9
10 > @(first @(mon futur (natpairc semi) (list 5 -1)))
11 ⇒5
```

These examples demonstrate how we may freely mix strategies for a variety of monitoring behaviors:

- Example 1 (lines 1-2) eagerly enforces the entire contract and, even though the second element of the pair is not part of the program's final result, the **eager**

[4]If the contract and expression do not contain side effects, the **semi** and **futur** strategies are operationally equivalent.

[5]We take a pair, here, to mean a list of two elements.

[6]We could have taken two strategies for each of the two subcontracts, but stick with a single one for simplicity of presentation

verification mechanism enforces natc on -1, signaling a violation.

- Example 2 (lines 4-5) eagerly enforces a pair of semi-eager monitors, yielding a pair of delayed expression (as (<**delay** ...> <**delay** ...>)). We use **first** to select the first element (a delayed expression) and **@** to retrieve the result (5).
- Example 3 (lines 7-8) creates a computational future that eagerly ensures both elements of the pair are natural numbers and, when the future's result is retrieved (via **@**), the monitor reports the violation.
- Finally, example 4 (lines 10-11) creates a parallel process that semi-eagerly monitors the two subcontracts. When we force both the future and the resultant delay, we receive the first element (5) without enforcing natc on -1.

Determining when values require forcing may be cumbersome in mixed-strategy settings, and thus MCON also includes the **extract** operator, which forces the top-level **delay** or **future**, if it exists:

```
> (let [x (mon semi natc 5)
        y (mon eager natc 5)]
    (+ (extract x) (extract y)))
⇒10
```

For example, consider a recursive definition of factorial that may or may receive contracted input:

```
(defn fact
  [x]
  (let [n (extract x)]
    (if (zero? n) 1 (* n (fact (- n 1))))))

> (+ (fact (mon semi natc 5)) (fact 5))
⇒240
```

If we wish to ensure that its input is a natural number, we may use func, the function contract combinator, to define the contract as:

```
(defn natfunc [strat] (func strat natc eager natc))
```

The func combinator creates *function contracts*, which consists of two or more subcontracts: a *pre-condition* (or list therein) to check on each function argument (natc in this example) and a *post-condition* to enforce on the function's output (also natc in this case). In this case, we parameterize the contract by its pre-condition subcontract, dictating how to enforce natc on the contracted function's input, and fix the post-condition strategy (dictating how to enforce natc on the output). Monitoring this contract on fact demonstrate the utility of **extract**: we may use any instantiation of natfunc.

```
(def fact-ee (mon eager (natfunc eager) fact))

> (fact-ee 5)
⇒120

(def fact-se (mon eager (natfunc semi) fact))

> (fact-se 5)
⇒120
```

While sprinkling **extract** around functions may seem cumbersome, the other alternative are either overloading the evaluator itself to handle these forms (which requires modifying Clojure's runtime with constructs such as Racket's chaperones and impersonators [15, 27]) or requiring programmers to extract *every* result [28].

2.2 New Verification Strategies (The Extensible)

When enforcing a function contract with func, we follow Findler and Felleisen [14], returning a new procedure. This procedures wraps the monitored procedure and, when run, ensures its input satisfies the precondition, applies the original function, and ensures that its output satisfies the post-condition, raising an error when either is violated. For example:

```
> (mon eager (natfunc eager) fact)
⇒(fn [x] (mon eager natc (fact (mon eager natc x))))
```

This is not, however, the only way to check if a function adheres to its contract. We may, instead, spot-check the function, wherein we use an *input generator* to ensure the function's contract holds for some number of generated inputs [9, 13]. To support such a verification technique, we define a new verification strategy, **spot**, in MCON via user code (using test.check's generator library as gen):

```
(defn spot-check
  [generator]
  (fn [contract dval]
    (let [f @dval]
      (if (not (fn? f))
          (throw (Exception. (str f " is not a function"))))
      (let [c (contract f)]
        (doall (map (fn [x] (c x)) (gen/sample generator 20)))
        f))))

(defn spot [g] (Strategy. "spot" (spot-check g)))
```

The Strategy record expects a strategy name and an *enforcement implementation*, defined here as the function returned from (spot-checkg). The idea is that, given a generator, we take 20 elements from it and apply the contracted function to each input. If the contract produces a violation, it will be reported; otherwise, we return the original function as the contract result. Using **spot** and Clojure's test.check natural number generator as gen/nat, we may spot-check functions:

```
(def nat-fun/eagerc (natfunc eager))

> (mon (spot gen/nat) nat-fun/eagerc (fn [x] (+ x 1)))
⇒<... function ...>

> (mon (spot gen/nat) nat-fun/eagerc (fn [x] (- x 10)))
⇒Contract Violation: output `-8` violated `natc`
```

This approach to defining new strategies allows programmers to write customized enforcement systems using the unified, extensible features of MCON.

2.3 Verification Meta-strategies (The Meta)

We are not limited to altering the monitoring behavior itself: we may, instead, supplement this behavior with a strategy-parameterized strategy. These *meta-strategies* take a strategy and produce a new strategy, supplementing the substrategy's behavior.

For example, the meta-strategy `with` takes a procedure as an argument and applies this procedure to the monitored value before returning the monitored value to the initiating process. To illustrate this behavior consider using `with` and println to print each value argument to a procedure:

```
(def add-with-print
  (mon eager
       (func (with println eager) natc
             eager natc))
  (fn [x] (+ x 10)))

> (add-with-print 10)
⇒"10"
   20
```

The `with` meta-strategy is also a key component in general runtime instrumentation; for example, instead of printing each argument to a terminal, we can log them in a mutable reference:

```
(def args (ref (list)))

(defn teller
  [x]
  (dosync (ref-set args (cons x (deref args)))))

(def add-with-tell
  (mon eager
       (func (with teller eager) anyc eager anyc)
       (fn [x] (+ x 1))))

> (add-with-tell 10)
11
> (add-with-tell 5)
6
> (add-with-tell 0)
1
> (deref args)
(0 5 10)
```

We may also extend this approach to general function profiling: each pre-condition and post-condition can report the current time, and the programmer can discover how much of their program runtime is spent in that particular procedure (including procedures it calls). This implementation, given in Figure 1, follows from the previous example: we instrument the pre- and post-conditions via `with` to store the current system time, tagged with either `:pre` or `:post`, in a global reference. At the end of the program, we can get an entire list of these calls and compute, e.g., average call time or program percentage.

This style of strategy and meta-strategy dispatch gives programmers an extensible, uniform interface for general runtime inspection of program values and their properties. We will examine more advanced examples after we explain

```
(def times (ref (list)))

(defn cur-time[] (System/currentTimeMillis))

(defn tag-timer
  [tag]
  (dosync (ref-set times
                   (cons (list tag (cur-time))
                   (deref times)))))

(defn slow-fact
  [n]
  (cond
    (zero? n) 1
    :else (dosync (Thread/sleep (* n 10))
                  (* n (fact (- n 1))))))

(def slow-fact-timed
  (mon eager
       (func (with (fn [x] (tag-timer :pre)) eager) natc
             (with (fn [x] (tag-timer :post)) eager) natc)
       slow-fact))

> (slow-fact-timed 10)
⇒3628800
> (deref times)
⇒((:post 1493529262817) (:pre 1493529262254))
```

Figure 1. An implementation of function-level runtime profiling in MCON.

our approach in detail. Furthermore, we may perform this instrumentation *in tandem* with monitoring standard pre- and post-condition contracts on the function: in Figure 1, we ensure each input and output is a natural number while recording timing information.

3 A Fistful of Strategies

We now present an implementation of the system outlined in the previous section, organized as follows: first, we introduce the overarching checking structures, including strategies, meta-strategies, and `mon`; next, we define a number of contract combinators and strategies; and, finally, we define our meta-strategies. Our implementation is presented in Figures 2, 3, and 4.

3.1 The MCON Framework

Our core framework is intentionally spartan: following the work presented by Swords et al. [28], our only requirements are (1) that we can change enforcement behavior based on the strategy, and (2) that the contracted expression is not evaluated "too soon." This core is defined in Figure 2.

We meet the first requirement by pushing each strategy and meta-strategy into records that include their own enforcement implementation, and thus behavior is implicitly strategy-defined: the mon-meta and mon-flat definitions retrieve the `:impl` field from the strategy and blindly use it to

```
(defrecord Strategy [sname impl])

(defrecord Metastrat [sname impl substrat])

(defn mon-flat
  [strat contract dval]
  (cond
    (instance? Strategy strat)
      ((:impl strat) contract dval)
    :else (Exception. (str "Invalid strategy: " strat "\n"
                           " contract: " contract "\n"
                           " input:" dval "\n"))))

(defn mon-meta
  [strat contract dval]
  (cond
    (instance? Metastrat strat)
      ((:impl strat) contract dval (:substrat strat))
    (instance? Strategy strat)
      (mon-flat strat contract dval)
    :else (Exception. (str "Invalid strategy: " strat "\n"
                           " contract: " contract "\n"
                           " input:" dval "\n"))))

(defmacro mon
  "Check a contract with a specific strategy"
  [strat contract value]
  `(mon-meta ~strat ~contract (delay ~value)))

(defn extract
  [exp] (if (or (delay? exp) (future? exp)) @exp exp))
```

Figure 2. Monitoring macro and strategy and meta-strategy records for MCON.

```
(defn predc
  [f]
  (fn [x]
    (if (f x)
        x
        (throw (Exception. (str "Contract violation: " x
                                " violated " f))))))

(defn con-ravel
  [args ins]
  (if (empty? ins)
      (list)
      (cons (concat (take 2 args) (list (first ins)))
            (con-ravel (drop 2 args) (rest ins)))))

(defn mon-fun
  [x] (mon (first x) (second x) (second (rest x))))

(defn func
  [& scs]
  (fn [f]
    (fn [& ins]
      (let [l (* 2 (count ins))
            cl (count scs)]
        (if (not (= (+ 2 l) cl)) (throw (Exception. ...))
        (let [posts (drop l scs)]
          (mon (first posts)
               (second posts)
               (apply f
                      (map mon-fun(con-ravel scs ins)))))))))
```

Figure 3. Contract combinators predc and func.

enforce the contract. As we have already seen, this modularization also allows programmers to write their own enforcement strategies that will work seamlessly within the existing framework.

We meet our second requirement via **defmacro** and **delay** in our definition of **mon**: the **mon** macro immediately delays the monitored expression, allowing each strategy to evaluate the expression if and when it becomes necessary. Each base strategy takes responsibility for evaluating this delayed term.

The rest of MCON is built as "user" programs on top of this framework.

Contract Combinators. Using this core definition, we may now define contract combinators, including predc and func (the latter of which takes extra machinery in order to unravel and correctly group its variable arguments), both given in Figure 3. In MCON, contracts are functions that perform their own enforcement, taking the monitored value and performing the necessary computation for verification. To this end, predc takes a predicate, applies it to its input, and either returns the input or raises an error.

Throwing exceptions to signal contract violations, however, is atypical of combinators: aside from predicate contracts, contract combinators (including func in Figure 3) are typically recursively defined in terms of **mon**, where the predicate contracts at the leaves will signal violations. Even so, contracts are always treated as functions from the monitored term to the monitor result, allowing strategies to proceed without considering contract shape and allowing users to define new contract combinators as user-level procedures.

3.2 Defining Contracts and Strategies

Next, we define strategies including skip, eager, semi-eager, future-based, concurrent, and spot-checking base strategies for MCON (omitting further discussion of **spot**), all given in Figure 4.

Skip Verification. The **skip** verifications strategy *skips* enforcement, forgoing contract enforcement. This strategy finds uses in production systems (e.g., developing with contracts and turning them off for deployment) and as a stand-in when the strategy, not the contract, is performing the desired result (e.g., we could replace **eager** with **skip** in Figure 1). To implement **skip**, we evaluate the monitored expression (discarding the **delay** that **mon** applies) and return the result.

Eager Verification. The **eager** verification strategy follows the original presentations of contracts by Meyer [23]: it suspends the user evaluator, completely enforcing the contract, and then either resumes the user evaluator with the monitor

```
;; Verification Strategies

; Skip Verification
(defn skip-check
  [contract dval]
  @dval)

(def skip (Strategy. "skip" skip-check))

; Eager Verification
(defn eager-check
  [contract dval]
  (contract @dval))

(def eager (Strategy. "eager" eager-check))

; Semi Verification
(defn semi-check
  [contract dval]
  (delay (contract @dval)))

(def semi (Strategy. "semi-eager" semi-check))

; Future Verification
(defn future-check
  [contract dval]
  (future (contract @dval)))

(def futur (Strategy. "future" future-check))

; Conc Verification
(defn conc-check
  [contract dval]
  (do (a/go (contract @dval)) @dval))

(def conc (Strategy. "conc" conc-check))

; Spot-Checking Verification (functions only)
(defn spot-check
  [generator]
  (fn [contract dval]
    (let [f @dval]
      (if (not (fn? f))
          (throw (Exception. (str f " is not a function"))))
      (let [c (contract f)]
        (doall (map (fn [x] (c x)) (gen/sample generator 20)))
        f))))

(defn spot [g] (Strategy. "spot" (spot-check g)))
```

```
;; Verification Meta-Strategies

; With-Operator Verification
(defn with-check
  [fun]
  (fn [contract dval sub-strat]
    (let [res (mon sub-strat contract @dval)]
      (do (fun res)
          res))))

(defn with
  [fun strat]
  (Metastrat. "with" (with-check fun) strat))

; Random Verification
(defn random-check
  [rate]
  (fn [contract dval sub-strat]
    (if (< (rand) rate)
        (mon sub-strat contract @dval)
        @dval)))

(defn random
  [rate strat]
  (Metastrat. "rand" (random-check rate) strat))

; Communicating Verification
(defn comm-check
  [channel fun]
  (fn [contract dval sub-strat]
    (let [res (mon sub-strat contract @dval)]
      (do (a/put! channel (fun res))
          res))))

(defn comm
  [chan fun strat]
  (Metastrat. "comm" (comm-check chan fun) strat))

; Memoizing Verification
(defn memo-check
  [contract dval sub-strat]
  (mon sub-strat (memoize contract) @dval))

(defn memo
  [strat]
  (Metastrat. "memo" memo-checkstrat))
```

Figure 4. Monitoring strategy and meta-strategy implementations in MCON.

result (when the expression *satisfies* the contract) or raises an error (when the expression *violates* it):

```
> (+ 5 (mon eager natc -1))
⇒Contract Violation: `-1` violated `natc`
```

Since contracts are functions that perform their own enforcement, we implement **eager** by evaluating the monitored expression and applying the contract to it, allowing the contract to proceed with enforcement.

Semi-Eager Verification. The **semi** verification strategy, originally presented by Hinze et al. [18] and extended by others [3, 5, 6], was inspired by lazy evaluation, where computations are not performed until they are required by the program output. To mimic this behavior in a call-by-value language, the **semi** strategy suspends the monitor, returning a delayed expression to the user. To implement this behavior,

we follow our implementation of eager, wrapping the check expression in a delay:

```
> (mon semi natc 5)
⇒<delay ... (natc 5) ...>

> (let [x (mon semi natc -1)]
    (+ (fact 5) @x))
⇒Contract Violation: `-1` violated `natc`
```

When the program forces the delayed expression (via Clojure's @ operator), the contract is enforced and evaluation proceeds as with eager verification.

Future-Based Verification. The futur verification strategy, based on *Future Contracts* [10] and precisely defined by Swords et al. [28], creates a computational future to perform contract verification. In Clojure, we use the future keyword, wrapping it around an application of the contract to the monitored term, to utilize Clojure's runtime (i.e., the Java Virtual Machine) for parallel contract enforcement [17]:

```
> (mon futur natc 5)
⇒<future ... (natc 5) ...>

> (let [x (mon futur natc -1)]
    (+ (fact 5) @x))
⇒Contract Violation: `-1` violated `natc`
```

As with semi-eager verification, the resultant value may be retrieved with the @ operator, signaling errors at that time.

Concurrent Verification. Finally, concurrent verification is similar to *asynchronous verification* described by Swords et al. [28], concurrent verification performs "best-effort" enforcement, wherein remaining verifiers are discarded when the user program terminates. Unlike asynchronous verification, however, conc verification will never terminate the entire program in the event of an error: in Clojure, the monitoring process signaling an error does not stop other processes from running:

```
> (let [x (mon conc natc -1)]
    (+ x x))
⇒Contract Violation: `-1` violated `natc`
  -2
; or ⇒-2 without an error
```

This strategy is ideal for checking "soft" properties, where the user may or may not care about the result and is not willing to interrupt the overall program in the event of a violation. Furthermore, conc may be used to recreate the lazy tree fullness example in Swords et al. [28].

3.3 Meta-strategy Implementation

Finally, we turn our attention to implementing *meta-strategies*, revisiting with and introducing comm, random, and memo, which each take a sub-strategy and modify the behavior in some way. All of our meta-strategies are defined in Figure 4.

The with Meta-strategy. The with meta-strategy is the simplest: it takes a sub-strategy and a procedure and, similar to a dependent contract [14], it applies the procedure to the contracted result. Unlike a dependent contract, however, the meta-strategy applies the function and discards the result (instead of using the result as a new contact). We have already explored its utility and so focus on its implementation. Enforcement under the with meta-strategy proceeds as: the meta-strategy checks its contract using the sub-strategy in the :substrat field; the contracted result is passed as an argument to the provided procedure; and, finally, the contracted result is returned to the user process.

In our richer-blame variant, we adopt *indy*-style blame for this procedure: if the provided procedure misuses the value (e.g., if res is a contracted function that is given invalid input), the meta-strategy party is blamed for the contract violation.

The comm Meta-strategy. The comm, or *communicating*, meta-strategy is a special-case variation of the with meta-strategy that provides inter-process communication with the goal of simplifying the implementation of side-channel contracts [28]. It takes as input a strategy, a procedure f, and a *communication channel* i [4, 26]. After contract enforcement, the result is given to f, and the new result is written across the provided channel, and the contracted value is returned. We utilize this strategy in §4 to implement a refined function timer by modeling the timing report manager as a logging effect handler [20].

The random Meta-strategy. The random meta-strategy manifests the idea of *random enforcement*: a contract may or may not be checked at each enforcement site (i.e., occasionally ensuring that a function behaves correctly). For example, consider bst-insert, which takes a value and tree and inserts the value into the tree using a binary-search tree insertion algorithm: it is impractical to *always* ensure that the result is a binary-search tree because each check must iterate the entire tree, thus violating the asymptotic guarantee of binary-search tree insertion. As an alternative, we may use random to enforce the contract one-tenth of the time:

```
(def bst-ins
  (mon eager (func eager natc eager anyc
                   (random 0.1 eager) (bstc eager))
       bst-insert))
```

The function's post-condition, bstc eager, is enforced via (random 0.1 eager), which dictates that the contract should be enforced eagerly one-tenth of the time.

We directly implement random by generating a random number and testing it against the given rate: if the random number is less than the rate, we proceed with enforcement using the given sub-strategy; if not, we return the (forced) monitored expression.

The memo Meta-strategy. The last meta-strategy we present is memo, which memoizes a contract using Clojure's memoize operation before applying it. As with random, this is particularly effective for reducing performance overhead for contracted functions. For example, the following code will

check each input and output to fact exactly once, memoizing the results:

```
(def fact-m (mon eager (func (memo eager) natc
                             (memo eager) natc)
                  fact))
```

Implementation is straight-forward: we invoke Clojure's memoize operator on the contract, retrieving a memoized version of the procedure, and proceed with enforcement using this memoized contract.

4 For a Few Monitors More

With MCON in place, we turn our attention to a number of in-depth examples to demonstrate its utility and illustrate how multi-strategy monitoring, when combined with meta-strategies, yields a rich verification and instrumentation system. We present the following case studies: a recursive binary-search tree contract parameterized by its recursive and node-level enforcement (§4.1); a transition-based meta-strategy that uses a global reference to perform state machine runtime verification (similar to Racket's protocol contracts [9]) (§4.2); and a concurrency-based approach to general logging and instrumentation (§4.3).

4.1 Multi-Strategy Monitors

The multi-strategy approach to contract verification allows users to flexibly reuse contracts to yield a number of behaviors. Swords et al. [28] use the example of ensuring that a binary tree is a *binary-search tree*, and we recreate it in MCON, including the relevant combinator and contract. In Figure 5, we define a record to represent a tree node, a dependent tree contract combinator treedc (wherein the recursive contracts receive the current tree node's value as input [15]), and finally, define bstc, a tree contract parameterized by two strategies: rec-strat, which determines how to recursively enforce bstc on subtrees, and value-strat, which determines how to enforce the *value* contract on each value in the tree.

Using this contract, we may eagerly ensure that a binary tree is, indeed, a binary-search tree, and, moreover, we may vary the strategy to alter the performance characteristics and guarantees. Using **eager** for both strategies, for example, will *completely* traverse the tree, checking each node (an $O(n)$ operation):

```
> (mon eager (bstc eager eager) tree)
⇒<tree ...>
```

Although this ensures the tree is a binary-search tree at each node, it may be unsuitable as a contract for a binary-search tree insert function (which should have performance $O(\log n)$).

We may, instead, use **futur** at the top level of enforcement, creating a computational future that performs a concurrent $O(n)$ traversal and contract enforcement of the tree:

```
> (mon futur (bstc eager eager) tree)
⇒<future ...>
```

```
(defrecord BinTree [val left right])

(defn treedc
  "Dependent tree contract combinator"
  [sleaf cleaf sval cval srec cleft cright]
  (fn [tree]
    (if (nil? tree)
        (mon sleaf cleaf tree)
        (let [v (:val tree)]
          (BinTree.
            (mon sval cval v)
            (mon srec (cleft v) (:left tree))
            (mon srec (cright v) (:right tree)))))))

(defn bst-range [lo hi] (fn [v] (and (≥ v lo) (≤ v hi))))

(defn bstc
  "Binary search tree contract"
  [rec-strat value-strat]
  (letfn [(bstc [lo hi]
            (treedc
              eager anyc
              value-strat (predc (bst-range lo hi))
              rec-strat
              (fn [v] (bstc lo (extract v)))
              (fn [v] (bstc (extract v) hi))))]
    (bstc Integer/MIN_VALUE Integer/MAX_VALUE)))
```

Figure 5. A strategy-parameterized contract for ensuring a binary tree is a binary-search tree, using the dependent tree contract combinator treedc. Leaves are considered to be **nil** values.

Moreover, this parameterization allows us to change the contract behavior along two axes. First, consider enforcing bstc with **semi** as the recursive strategy and **eager** as the value strategy on tree, which is *not* a binary-search tree:

```
(def tree (BinTree. 5 (BinTree. 6 nil nil)
                      (BinTree. 7 nil nil)))

> @(:left @(mon semi (bstc semi eager) tree))
⇒Contract Violation: `6` violated `bst-range`
```

In this invocation, we force the resultant delayed contract. As the evaluator recursively enforces the contract, it eagerly enforces the contract (bst-range Integer/MIN_VALUE 5) on 6, signaling an error. This style of enforcement preserves the algorithm complexity for the monitored function: only observed nodes have their value contracts enforced, and so, for example, an $O(\log n)$ traversal of the tree will only check $\log n$ values. If, however, we use **semi** for the value contracts, we get even more conservative enforcement:

```
> (:val @(:left @(mon semi (bstc semi semi) tree)))
⇒<delay ...>

> @(:val @(:left @(mon semi (bstc semi semi) tree)))
⇒Contract Violation: `6` violated `bst-range`
```

Instead of enforcing each value contract as the node is forced, the monitor returns a BinTree node that contains a delayed value assertion (and two delayed sub-trees).

In MCON, you may write a contract once, and reuse it multiple times with myriad enforcement behaviors: bstc may also be used with spot-checking, concurrent checking, enforcement with logging (via meta-strategies), or any other strategy.

4.2 State Machine Runtime Verification

Our second in-depth example deals with verifying state-based program operation, similar to the *protocols* described by Dimoulas et al. [9], wherein a series of contracts collaborate to ensure a program is proceeding correctly. In lieu of analyzing a card game with complex rules, we demonstrate how to verify that a program does not request an iterator's next value without first checking it has one [19].

Transition Meta-Strategy System. We first introduce the **transition** and **transition-as** meta-strategies, given in Figure 6: in order to implement state machine runtime verification, we must track the current machine state (using a Clojure **ref**) and how it changes over the course of a program (using two meta-strategies that interact with the reference):

- The **transition** meta-strategy takes the strategy state (i.e., the reference to the current state), a from-state indicating the transition source, a to-state transition target, and a substrat sub-strategy to enforce the contract. This implementation is similar to the PROTOCOL-α form presented by Dimoulas et al. [9], allowing us to optionally model non-deterministic finite-state machines as

  ```
  (transition state :from (list :to-a :to-b) sub-strat)
  ```

  The to-state represents a *list* of valid next states, which is stored in the "machine state", and a valid transition is one in which the set of current machine state contains the **:from** state; if the state list does not include the **:from** state, the meta-strategy signals an error.
- The **transition-as** meta-strategy takes the strategy state and a procedure transition-fn parameterized by the current strategy state and the contracted value (similar to **with**). If the transition-fn returns the state **:error**, the meta-strategy signals an error; otherwise, the state is updated and evaluation proceeds. This allows the transition function to examine the *contract input* (similar to a dependent contract [14]), parameterizing its transition by the values currently flowing through the program.

We also provide make-contract-state to create a contract state in order to maintain implementation-agnostic behavior.

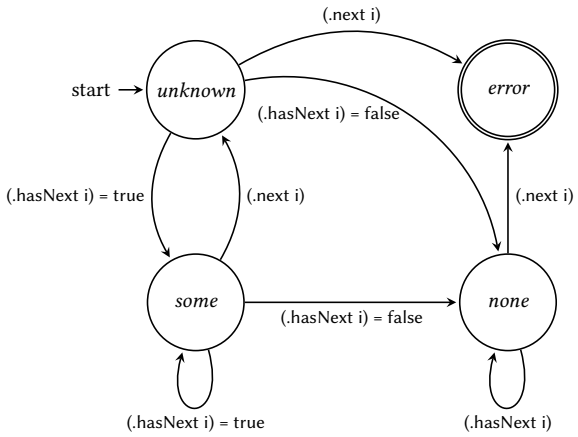
State-Transition Contracts. We can now use this transition-style meta-strategy to ensure correct usage behavior for iterators originating from Java libraries: we would like to ensure that, given an Java iterator, the programmer *always* calls hasNext on the iterator (and it returns true) before they call next on it. This state machine is given in Figure 7.

```
(defn make-contract-state
  [start-state ]
  (ref (list start-state )))

(defn in? [coll elm] (some #(= elm %) coll ))

(defn as-list [ls ] (if (list ? ls ) ls (list ls )))

(defn transition-check
  [state-ref from-state to-state ]
  (fn [contract dval sub-strat ]
    (let [res (mon sub-strat contract @dval)]
      (if (in? (deref state-ref ) from-state )
          (dosync (ref-set state-ref (as-list to-state ))
                  res )
          (throw (Exception. ...))))))

(defn transition
  [state-ref from-state to-state strat ]
  (Metastrat. (str "transition " from-state to-state )
              (transition-check state-ref from-state to-state )
              strat ))

(defn transition-as-check
  [state-ref transition-fn ]
  (fn [contract dval sub-strat ]
    (let [res (mon sub-strat contract @dval)
          to-state (transition-fn (deref state-ref ) res)]
      (if (not (= to-state :error))
          (dosync (ref-set state-ref (as-list to-state ))
                  res )
          (throw (Exception. ...))))))

(defn transition-as
  [state-ref transition-fn strat ]
  (Metastrat. (str "transition -as")
              (transition-as-check state-ref transition-fn )
              strat ))
```

Figure 6. State transition verification as meta-strategies. We have omitted the Exception messages due to length.

In order to model this behavior, we treat errors as contract errors (instead of an explicit state), and write monitors that model and enforce this behavior:

- We model .next's behavior using **transition**—any time the current state is not **:some** and we call .next on the iterator, we should produce an error; if the current state is **:some**, we should transition to the **:unknown** state.
- We model .hasNext's behavior based on its result (that is, as a flow-dependent transition based on .hasNext's output): after we run .hasNext, we use **transition-as** to inspect the current state *and* the result to transition across the appropriate lines in the diagram.




```
(def iter-state (make-contract-state :unknown))

(def next (mon eager (func eager anyc
                           (transition iter-state
                                       :some
                                       :unknown eager)
                           anyc)
                #(.next %)))

(defn hasNextTrans
  [con-result cur-state]
  (if (extract con-result) :some :none))

(def hasNext (mon eager (func eager anyc
                              (transition-as iter-state
                                             hasNextTrans
                                             eager)
                              anyc)
                  #(.hasNext %)))
```

Figure 7. A flow-dependent finite-state machine and its MCON implementation.

This implementation is given in Figure 7, and is used as:

```
;; Example 1
> (let [iter (.iterator
              (.keySet (java.lang.System/getProperties)))]
  (while (hasNext iter)
    (println (next iter))))
⇒...

;; Example 2
> (let [iter (.iterator
              (.keySet (java.lang.System/getProperties)))]
  (println (next iter)))
⇒Program performed invalid operation:
  Current state: (:unknown) Transition: :some -> :unknown
```

In the first example, the program terminates as expected. In the second, however, the program invokes next on the iterator before invoking hasNext, an invalid state transition, and the transition-enforcement mechanism raises an error.

Alternatives to the Transition Meta-strategy. We include the transition-style runtime system here as a secondary set of definitions because, as we demonstrate next, it is possible to implement a similar set of behaviors using core.async and the comm meta-strategy.

```
(defn timer-task
  [in-chan out-chan timer-info]
  (let [action (a/<!! in-chan)]
    (cond
      (= action :result)
        (let [res (filter #(= (first %) :time) timer-info)]
          (a/>!! out-chan
                 (/ (reduce + (map second res))
                    (float (count res)))))
      (and (list? action) (= (first action) :pre))
        (timer-task
          in-chan out-chan
          (cons (list :pre (second action)) timer-info))
      (and (list? action) (= (first action) :post))
        (timer-task
          in-chan out-chan
          (cons (list :time
                      (- (second action)
                         (second (first timer-info))))
                (rest timer-info)))
      :else
        (timer-task in-chan out-chan timer-info))))

(defn start-timer-task
  [in-chan out-chan]
  (a/go (timer-task in-chan out-chan (list))))

(defn timer-func
  [in-chan out-chan]
  (do
    (start-timer-task in-chan out-chan)
    (func (comm in-chan
                (fn [_] (list :pre (cur-time)))
                eager)
          anyc
          (comm in-chan
                (fn [_] (list :post (cur-time)))
                eager)
          anyc)))
```

Figure 8. Function timing with a concurrent process.

4.3 Function Timing with Concurrent Logging

Our final in-depth example revisits using meta-strategies as timers to showcase how the communication meta-strategy may be used to model general runtime instrumentation: instead of using with and an ad-hoc reference, we may implement this behavior using comm and a concurrent process that acts as a "timing report manager." This implementation is given in Figure 8, where Clojure's core.async library is named a and imported. First, we define a timer-task, which is the main loop of the concurrent timer process. This concurrent process will take a *tagged* message indicating which action to take (storing a function start time or end time, end time, retrieving the overall timing result, or terminating the timing process), and maintains a list of execution times (by

storing each pre-condition enforcement time-stamp and computing its difference from the post-condition enforcement time-stamp). We provide start-timer-task for ease of use in creating the new process.

After that, we define the function contract timer-func which starts a timing task and uses **comm** to set up pre-condition and post-condition contracts that write a TimerAction tagged with the appropriate position (**:pre** or **:post**) and the current time across the appropriate channel. We use this as:

```
> (let [in  (a/chan)
        out (a/chan)
        f (mon eager (timer-func in  out)  slow-fact)]
    (f  15)
    (f  5)
    (a/put!  in  :result)
    (a/<!!  out))
⇒104.33
```

We get two channels for input and output, monitor our previous slow factorial implementation, start the timer task, call the monitored procedure a number of times, then retrieve the timing result (and terminating the timer process).

While this timer is a reimplementation of the code in Figure 1, this approach illustrates a more general result: this encoding corresponds directly to algebraic effects as sessions, and *any* logging-style effect that may be expressed as an algebraic-effect-style processes may interact with contracts via **comm**, including the state transition monitor presented in §4.2 and other state-interaction effects [1, 2, 20, 22, 25]. The multi- and meta-strategy approach in MCON provides a generalized interaction system between these types of effects and contract monitors.

5 Related Works

Eiffel [23] first introduced the idea of software contracts and the idea of writing programs with ubiquitous contracts, relying on eager contract verification. Findler and Felleisen [14] introduced contracts to the functional world, and also presented a variation of semi-eager verification as part of their module system. Since then, a number of other works have introduced additional enforcement mechanisms, including semi-eager enforcement [5, 18], parallel enforcement [10], lazy-style enforcement [5, 15], contracts tracing module interactions [12], option contracts [8], and access control contracts for security [24]. We have recreated several of these in MCON, and conjecture that the others may be expressed as extensions to our framework. The access control mechanism presented by Moore et al. [24], for example, utilizes contracts with an extra keyword parameter that indicates additional authentication requirements to the underlying contract, and it seems possible to create an auth metea-strategy that replicates some or all of this behavior (assuming Clojure's runtime provides the necessary introspection primitives).

Dimoulas et al. [9] observe and explore the usage of contracts to implement domain-specific runtime verification systems in Racket, including spot-checking and transition-enforcement systems, that are similar in nature to our **spot** and **transition** systems. We conjecture that their other domain-specific systems, including specialized interfaces type enforcement, may also be adapted to MCON.

Enforcement strategies have been surveyed by Degen et al. [5], Dimoulas et al. [11], and Swords et al. [28], each along different axes and criteria. Our work eschews classification for focus on implementation, which loosely follows the Racket library of Swords et al. [28], who introduce strategy-parameterized contracts and outline a number of strategy implementations (including many of those in MCON), using communicating processes to model contract interactions with the underlying user program. Our primary deviations in implementation are, first, that we avoid processes and parallelism whenever possible to reduce contract enforcement overhead and, second that we introduce the notion of meta-strategies. We conjecture that the meta-strategy framework may be described in their λ_{cc} calculus, giving rise to a formalism for meta-strategies as mediating processes between the user and contract evaluators, reminiscent of the algebraic effect layering presented by Kammar et al. [20].

Clojure's core.spec system uses the notion of "specifications" with different enforcement methods, including compile-time analysis and runtime checking. Unlike MCON, where a function may be enforced at each use or spot-checked, higher-order function inputs are *always* spot-checked in core.spec. Combining the multi- and meta-strategy approach with core.spec remains as future work.

Cartwright and Felleisen [2] describe using communication to model effects, and the recent rise of algebraic effect models [1, 20–22] and their close connection to effects-as-processes [25] have opened the door for managing algebraic effects as process interactions, which is a natural fit with the process-based contract verification framework described by Swords et al. [28], and, ultimately our runtime instrumentation examples. As far as we know, this is the first work to leverage this connection in the context of contract verification.

6 Conclusion

In this work, we present an account of a multi-strategy contract monitoring system originally presented by Swords et al. [28] and introduce the notion of *meta-strategies*, which modify base verification strategies to expose a rich set of enforcement behaviors. Some of our meta-strategies coincide with existing ad-hoc extensions to contract systems, allowing us to express them as parts of the verification process itself. The addition of meta-strategies to the core strategy calculus gives us the ability to verify advanced program properties for collaborative contracts and even explore using the monitoring interaction for general runtime instrumentation and effects.

Future Work. Future work includes implementing improved error messages, removing the need to explicitly pass blame and providing augmented versions of **def** and **defn** to automatically inject the information when possible; expressing *option contracts* as a meta-strategy [8], which may require chaperone-esque mechanisms [27]; and constructing a formal semantics for meta-strategies and exploring a proof of *complete monitoring* [11] in this context.

Acknowledgments

We would like to thank Rebecca Swords, Ambrose Bonnaire-Sergeant, Matthew Heimerdinger, Amr Sabry, Sam Tobin-Hoch-stadt, and Christian Wailes for their continuous feedback. This material is based upon work supported by the National Science Foundation under Grant No. 1117635 and Grant No. 1217454.

References

[1] Andrej Bauer and Matija Pretnar. 2012. Programming with Algebraic Effects and Handlers. (2012). arXiv:1203.1539 [cs.PL].

[2] Robert Cartwright and Matthias Felleisen. 1994. Extensible Denotational Language Specifications. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS '94)*. Springer-Verlag, London, UK, UK.

[3] Olaf Chitil. 2012. Practical Typed Lazy Contracts. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 10.

[4] Clojure. 2017. Clojure core.async. (2017). https://github.com/clojure/core.async

[5] Markus Degen, Peter Thiemann, and Stefan Wehr. 2009. True Lies: Lazy Contracts for Lazy Languages (Faithfulness is Better than Laziness). In *Arbeitstagung Programmiersprachen (ATPS)*. Springer, Lübeck, Germany.

[6] Markus Degen, Peter Thiemann, and Stefan Wehr. 2010. Eager and delayed contract monitoring for call-by-value and call-by-name evaluation. *The Journal of Logic and Algebraic Programming* 79 (2010).

[7] Christos Dimoulas and Matthias Felleisen. 2011. On Contract Satisfaction in a Higher-order World. *ACM Transactions on Programming Languages and Systems* 33, 5, Article 16 (Nov. 2011), 29 pages.

[8] Christos Dimoulas, Robert Bruce Findler, and Matthias Felleisen. 2013. Option Contracts. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 20.

[9] Christos Dimoulas, Max S. New, Robert Bruce Findler, and Matthias Felleisen. 2016. Oh Lord, Please Don't Let Contracts Be Misunderstood. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP '16)*.

[10] Christos Dimoulas, Riccardo Pucella, and Matthias Felleisen. 2009. Future Contracts. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP '09)*. ACM, New York, NY, USA, 12.

[11] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete monitors for behavioral contracts. In *Proceedings of the 21st European Symposium on Programming Languages and Systems*. Springer-Verlag, Berlin, Heidelberg.

[12] Tim Disney, Cormac Flanagan, and Jay McCarthy. 2011. Temporal Higher-order Contracts. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 13.

[13] Funda Ergün, Sampath Kannan, S. Ravi Kumar, Ronitt Rubinfeld, and Mahesh Viswanathan. 1998. Spot-checkers. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC '98)*. ACM, New York, NY, USA, 10.

[14] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-order Functions. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM, New York, NY, USA, 12.

[15] Robert Bruce Findler, Shu-Yu Guo, and Anne Rogers. 2008. Lazy Contract Checking for Immutable Data Structures. In *Implementation and Application of Functional Languages (IFL '08)*. Springer-Verlag, Berlin, Heidelberg, 18.

[16] Daniel Friedman and David Wise. 1976. *The Impact of Applicative Programming on Multiprocessing*. Technical Report 52. Indiana University, Computer Science Department.

[17] Daniel Higginbotham. 2015. *Clojure for the Brave and True: Learn the Ultimate Language and Become a Better Programmer*. No Starch Press.

[18] Ralf Hinze, Johan Jeuring, and Andres Löh. 2006. Typed Contracts for Functional Programming. In *Proceedings of the 8th International Conference on Functional and Logic Programming (FLOPS'06)*. Springer-Verlag, Berlin, Heidelberg, 18.

[19] Omar Javed, Yudi Zheng, Andrea Rosà, Haiyang Sun, and Walter Binder. 2016. Extended Code Coverage for AspectJ-Based Runtime Verification Tools. In *Runtime Verification: 16th International Conference (RV 2016)*, Yliès Falcone and César Sánchez (Eds.). Springer.

[20] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ICFP'13*.

[21] Daan Leijen. 2012. Koka: A Language with Row-Polymorphic Effect Inference. In *1st Workshop on Higher-Order Programming with Effects (HOPE 2012)*.

[22] Conor McBride. 2012. The Frank Manual. (2012). https://personal.cis.strath.ac.uk/conor.mcbride/pub/Frank/.

[23] Bertrand Meyer. 1992. *Eiffel: the language*. Prentice-Hall, Inc.

[24] Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. 2016. Extensible Access Control with Authorization Contracts. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 20.

[25] Dominic Orchard and Nobuko Yoshida. 2016. Effects As Sessions, Sessions As Effects. In *Proceedings of the 43rd Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA.

[26] John H. Reppy. 1993. Concurrent ML: Design, Application and Semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*. Springer-Verlag, London, UK, UK, 34.

[27] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. Chaperones and Impersonators: Run-time Support for Reasonable Interposition. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 20.

[28] Cameron Swords, Amr Sabry, and Sam Tobin-Hochstadt. 2015. Expressing Contract Monitors As Patterns of Communication. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 13.