Contract Systems Need Domain-Specific Notations

Cameron Moy
PLT @ Northeastern University, Boston, MA, USA

Ryan Jung[1]
Northeastern University, Boston, MA, USA

Matthias Felleisen
PLT @ Northeastern University, Boston, MA, USA

Abstract

Contract systems enable programmers to state specifications and have them enforced at run time. First-order contracts are expressed using ordinary code, while higher-order contracts are expressed using the notation familiar from type systems. Most interface descriptions, though, come with properties that involve not just assertions about single method calls, but entire call chains. Typical contract systems cannot express these specifications concisely. Such specifications demand domain-specific notations. In response, this paper proposes that contract systems abstract over the notation used for stating specifications. It presents an architecture for such a system, some illustrative examples, and an evaluation in terms of common notations from the literature.

2012 ACM Subject Classification Software and its engineering → Domain specific languages

Keywords and phrases software contracts, domain-specific languages

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.14

Category Pearls/Brave New Ideas

Funding This work was supported by NSF grant SHF 2116372.

1 From General-Purpose to Domain-Specific Notations

Eiffel's contract system assumes that programmers can formulate Boolean expressions to articulate simple pre-conditions and post-conditions for methods [33, 34]. Such expressions suffice to state constraints that quotidian type systems cannot express. Here is an example, from Eiffel's documentation, of a method that sets the `seconds` field of a clock object:

```
set_seconds (s: INTEGER)
  require
     valid_argument_for_seconds: 0 <= s and s <= 59
  do
     seconds := s
  end
```

By monitoring the pre-condition, Eiffel's contract system can signal an error when control enters the method, not after `seconds` has been set and retrieved elsewhere. Eiffel's syntax has `require` and `ensure` keywords that cleanly separate specifications from implementations.

Findler and Felleisen [17] generalize Eiffel's design to modern, higher-order languages with first-class functions and objects. They realize that such a generalization necessitates creating and applying wrappers around behavioral values. Here is the code for checking that a `derivative` function takes real-valued functions to real-valued functions:

[1] Currently working in industry.

© Cameron Moy, Ryan Jung, and Matthias Felleisen;
licensed under Creative Commons License CC-BY 4.0

```
(define (derivative unsafe-f)
  (define (check-real->real g)
    (lambda (x)
      (check-real (g (check-real x)))))
  (define f (check-real->real unsafe-f))
  (check-real->real (lambda (x) —— compute derivative ——)))
```

But programmers should never read, let alone write, such complex code. Contract systems, such as the one found in Racket [15, 20], therefore supply the familiar notation from type systems to eliminate these awkward and repetitive patterns:

```
(define/contract (derivative f)
  (-> (-> real? real?) (-> real? real?))
  (lambda (x) —— compute derivative ——))
```

In other words, developers may use the `->` notation as a shorthand for the tedious wrappers in the first version. In Eiffel, contract-specific notations are mere conveniences, but here the notation is essential—writing complex higher-order contracts without them is infeasible.

Moy and Felleisen [37] present an extension of higher-order contracts, dubbed *trace contracts*, that can express and monitor properties across multiple function and method calls. The idea looks straightforward: collect a trace of method-call events and check the trace with a functional predicate whenever it is updated.

Consider the HasNext property that Java programmers must respect when they use an iterator: each call to `next` must follow a call to `hasNext` [39]. A typical implementation would equip the class with an extra field that determines whether a violation has occurred:

```
class Iterator {
  private boolean canCallNext = false;

  public boolean hasNext() {
    this.canCallNext = true;
    —— method body ——
  }

  public Element next() {
    if (!canCallNext)
      throw new Exception("HasNext property violated");

    this.canCallNext = false;
    —— method body ——
  }
}
```

Comingling run-time checks with proper functionality is a technique known as *defensive programming*. As Meyer [33, 34] argued a long ago, comingling is bad for the person who maintains the implementation, bad for the person who programs to the interface, and bad for the person who wants to implement the interface differently.

By contrast, a contract system brings relief to all three groups. It enables the developer of an *interface* to state contracts for cross-cutting specifications and to attach it to concrete implementations. A contract, independent of any particular implementation, confers two critical advantages:

1. Developers can program to interfaces that express all constraints as code explicitly.

2. Contracts are compiled to run-time checks that monitor implementations of interfaces, so developers can trust that specification violations are reported in a timely fashion.

With trace contracts, the creator of an iterator class can express the HasNext property as a contract and gain all the benefits thereof:

```
(provide
  (contract-out [an-iterator-object has-next/c]))

(define has-next/c
  (object-trace/c
    #:satisfies (regex (star (sequence 'has-next (optional 'next))))
    [has-next (->m boolean?) 'has-next]
    [next     (->m any/c)    'next]))
```

— *definition of the iterator* —

This module exports an iterator object and, in doing so, attaches a contract that concisely states the HasNext property with the regular expression $(\texttt{hasNext}, \texttt{next?})^*$ over the alphabet of method names. Other than the usual fully-parenthesized syntax of Racket, the regular expression is rendered directly. The use of a problem-specific notation, here regular expressions, clarifies the developer's intent much better than comingled code, and it still permits the automated construction of dynamic checks. Each method, after the signature given by `->m`, is associated with an *event*. An event is a value to be added to the trace when that method is called. Here, the events are just atomic symbols: `'has-next` and `'next`.

Which brings us to a brave new idea: *contract systems should be abstracted over domain-specific notations for specifications*. In other words, it should be easy to plug in any number of domain-specific specification languages into a contract system. This paper illustrates the idea with *trace contracts* as the enforcement mechanism and a variety of *domain-specific specification languages*, which together yield domain-specific contracts (DSCs). The remainder of the paper describes DSCs and the underlying architecture in detail, shows how it works with a simple example (Section 2), presents an example for a sophisticated real-world API (Section 3), describes a general plan for implementing DSCs in different host languages (Section 4), and evaluates a Racket implementation of DSCs (Section 5).

2 Domain-Specific Contracts in Action

Many interface specifications employ temporal terminology to constrain sequences of method calls. Consider the MapIterator property from Java's API [40]. This property states that an `Iterator`, created from a `Collection`, which is itself derived from a `Map`, cannot be used *after* the `Map` has been mutated. The word "after" suggests a temporal relationship—a need to look back at whether a mutating method of `Map` has previously been called.

An interface, therefore, should use a matching formal language, such as temporal logic, to state the property. Given that such statements tend to look to the past, Past Linear Time Logic (PLTL) [5] is one suitable choice for stating the MapIterator contract. Indeed, the definition of satisfaction for PLTL formulas employs finite traces of past events—just like trace contracts.

PLTL comes with the primitive temporal operators $\bullet\,\phi$ (pronounced "previous") and $\phi\,\mathsf{S}\,\psi$ (pronounced "since"). Otherwise, PLTL looks just like first-order logic. All other temporal operators, for example $\blacklozenge\,\phi$ (pronounced "once"), are derived notions. With these

operators, an *invalid* sequence of method-call events is easy to express: `next` ∧ ♦ `mutate`. In prose, calling the `next` method of an iterator is invalid if the map has been mutated.

One contribution of this paper is a number of libraries, including one for PLTL, that developers can use in contracts. So, suppose a developer wishes to release a collections library that states and enforces the MapIterator property. Realizing this desire calls for three steps.

First, the developer imports the trace contract library and the PLTL library:

```
(require trace-contract)
(require logic/pltl)
```

Importing `trace-contract` brings in, among other forms, `object-trace/c`. As the introduction mentions, `object-trace/c` constructs an object contract that records a trace of method-call events. This trace is checked by a predicate every time a new element is added.

Second, the developer formulates and names the PLTL formula associated with the MapIterator property:

```
(define-pltl map-iterator-violation
  (∧ 'next (♦ 'mutate)))
```

Using the PLTL library the developer can formulate the predicate as the above temporal formula, matching the informal prose found in Java's documentation. To use a named PLTL formula, a programmer must use the `pltl` form, which translates it to a (state) `machine` representation. A satisfaction checker for `machine`s can check whether a trace satisfies or refutes a formula:

```
> (define s1 (list 'next 'next))
> (machine-accepts? (pltl map-iterator-violation) s1)
#false
> (define s2 (list 'next 'next 'mutate 'next))
> (machine-accepts? (pltl map-iterator-violation) s2)
#true
```

Third, the developer combines a PLTL expression with `object-trace/c` to create an enforceable contract. Figure 1 shows a first attempt at such a contract for MapIterator.

```
(define map-iterator/c
  (object-trace/c
    #:refutes (pltl map-iterator-violation)
    [next (->m any/c) 'next]))

(define map-collection/c
  (object-trace/c
    [iterator (->m map-iterator/c)]))

(define map/c
  (object-trace/c
    [keys   (->m map-collection/c)]
    [remove (->m any/c void?)       'mutate]
    [set    (->m any/c any/c void?) 'mutate]))
```

Figure 1 First Attempt: Checking the MapIterator Property

The three displayed definitions express a specification involving three methods across two classes, with an intervening class that bridges the gap. Two methods from the `Map`

class—`remove` and `set`—mutate the `Map`. The intent with these contracts is that any iterator derived from a collection of keys becomes unusable when the underlying map is mutated. The `#:refutes` clause requests that the PLTL formula named `map-iterator-violation` is translated into a predicate over the trace of method-call events. If this code ever discovers that this formula is satisfied, then the contract system signals a violation.

As is, these contracts are not quite right. An `object-trace/c` contract knows only about method calls for the current object. Therefore, the `map-iterator/c` contract sees only calls to the `next` method and is unaware of any mutations to the underlying map.

```
(define (map-iterator/c map/c)
  (object-trace/c
    #:refutes (pltl map-iterator-violation)
    #:include map/c
    [next (->m any/c) 'next]))

(define (map-collection/c map/c)
  (object-trace/c
    [iterator (->m (map-iterator map/c))]))

(define map/c
  (object-trace/c
    [keys   (->m (map-collection/c this-contract))]
    [remove (->m any/c void?)        'mutate]
    [set    (->m any/c any/c void?) 'mutate]))
```

Figure 2 Checking the MapIterator Property

Figure 2 shows how to adapt these contracts to inform the `map-iterator/c` contract of the relevant method calls. To express this relationship, the contracts for collection and iterator objects become *functions* that accept the `map/c` contract itself. The `object-trace/c` form uses the keyword `this-contract` which, similar to `this`, refers to the `object-trace/c` contract itself. While the `map-collection/c` function just passes this contract through to the `map-iterator/c` function, the latter actually makes use of it. The `#:include` clause informs the contract system that events from the trace of `map/c` are to be injected into the trace for `map-iterator/c`. Consequently, `map-iterator-violationn`, which refers to the `'mutate` event, now works correctly in the `#:refute` clause.

These contracts look simple, but the underlying property is quite sophisticated. Consider the scenario where an iterator is created, the underlying map is mutated, and then another iterator is created. It should be the case that the first iterator is invalidated, while the second one is usable. In other words these two iterators require independent pieces of state. Some events, such as mutations to the underlying map, are shared with both. Other events, such as calls to `next`, are particular to a specific iterator and are not shared. All this careful state management is handled automatically.

Conversely, manually constructing defensive checks to correctly keep track of this information is difficult. **??** shows an implementation of defensive checks for MapIterator, similar to the examples shown in Section 1. The code is complex and takes several pages to explain. By contrast, Figure 2 is nearly self-explanatory.

3 Real Code Needs a Mixture of Domain-Specific Notations

Interfaces should be expressed as directly as possible using the notation most appropriate for the property at hand. Sometimes a single interface benefits from mixing several domain-specific notations. The DSCs presented here are flexible enough to allow a heterogeneous mixture of notations because they all compile to the same uniform mechanism for capturing and checking traces of events.

This section presents a series of contracts that ensure TCP sockets are used correctly. Consider an idealized TCP interface, similar to the one found in Java, which consists of four classes: a TCP manager, a TCP listener, an input stream, and an output stream. A TCP manager, when given a port number, is responsible for creating a TCP listener that deals with the given port. The purpose of a TCP listener is to accept client connections. For each connected client, a TCP listener produces an input and output stream for receiving and sending data, respectively.

Many languages, including Java, guarantee the following properties of TCP sockets:

- StreamClose. Input and output streams can be used until either the stream itself it closed or the underlying TCP listener is closed.
- PortExclusivity. At any point within the program, there can be at most one active TCP listener on each port.

The Java standard library uses defensive programming to enforce these properties.

DSCs can nicely express both properties, but they warrant two different notations. The StreamClose property can be expressed as a regular expression:

```
(define-re port-re
  (seq (star `use) (opt (union `close-stream `(close-listener ,_)))))
```

This regular expression accepts traces of method-call events that use a stream (input or output) until it has been closed, or the underlying listener is closed. There is one unusual feature here, which is the close listener event. In all prior examples, events were symbolic names of methods carrying no additional data. The close listener event also carries the port number of the corresponding listener, which is simply ignored in `port-re` using a wildcard pattern. Why the close listener event carries a port number will be clarified momentarily.

The PortExclusivity property is not easily expressed as a regular expression. Another formalism, known as quantified event automata (QEA [4]), is a good match:

```
(define manager-qea
  (qea
   (∀ port)
   (start ready)
   [-> ready `(listen ,port) listening]
   [-> listening `(close-listener ,port) ready]))
```

A QEA is similar to an ordinary finite-state machine, but it is enriched with quantifiers. By quantifying over a variable, a QEA describes a family of automata: one automaton per instantiation of all quantified variables. Intuitively, `manager-qea` constructs a two-state automaton for every possible port number. These automata are created on demand, though, to avoid unnecessary overhead. Each automaton starts in the `ready` state and transitions to a `listening` state when a new TCP listener is created on that port. When that listener is closed, the automaton moves back to the `ready` state. Each listener event must carry a port number, so the event can be dispatched to the appropriate automaton.

```
(define (input-stream/c listener/c)
  (object-trace/c
   #:satisfies port-re
   #:include listener/c
   [read-byte (->m byte?) 'use]
   [peek-byte (->m byte?) 'use]
   [close     (->m void?) 'close-stream]))

(define (output-stream/c listener/c)
  (object-trace/c
   #:satisfies port-re
   #:include listener/c
   [write-byte (->m byte? void?) 'use]
   [close      (->m void?)       'close-stream]))

(define (tcp-listener/c manager/c port)
  (object-trace/c
   #:extend manager/c
   [close  (->m void?) '(close-listener ,port)]
   [accept (->m (values (input-stream/c this-contract)
                        (output-stream/c this-contract)))]))

(define tcp-manager/c
  (object-trace/c
   #:satisfies manager-qea
   [listen (->dm ([port (integer-in 0 65535)])
                 [res (tcp-listener/c this-contract port)])
           '(listen ,port)]))
```

Figure 3 The TCP Interface

Using `object-trace/c`, these logical specifications can be connected to the TCP interface itself; see Figure 3. Two differences from the example in Section 2 are worth pointing out. First, `tcp-manager/c` uses the dependent method contract form [48] `->dm` to produce an event that is dependent on the argument given to `listen`. This form permits naming the arguments and result, binding those names for use in the event constructor. Here, the listen event uses the `port` argument. Second, `tcp-listener/c` uses `#:extend` to contribute events to the parent `manager/c` contract. As shown in Figure 2, `#:include` allows events to flow from "ancestor" to "descendant" objects. The `#:extend` keyword allows events to flow in the other direction, from "descendant" to "ancestor" objects. Specifically, the TCP manager (i.e., parent object) must be aware if a listener (i.e., a child object) is closed on a particular port, because then that port becomes available for listening once again according to PortExclusivity. Using `#:extend manager/c` communicates the close listener events from the listener to the manager. It turns out that StreamClose requires information to flow in the other direction because input and output streams (i.e., children objects) must be closed if the underlying listener (i.e., parent object) is closed. This direction is covered using `#:include listener/c`. In sum, the example demonstrates that `object-trace/c` can accommodate information flow in both directions, even within the same interface, and can mix distinct notations to check all desired properties.

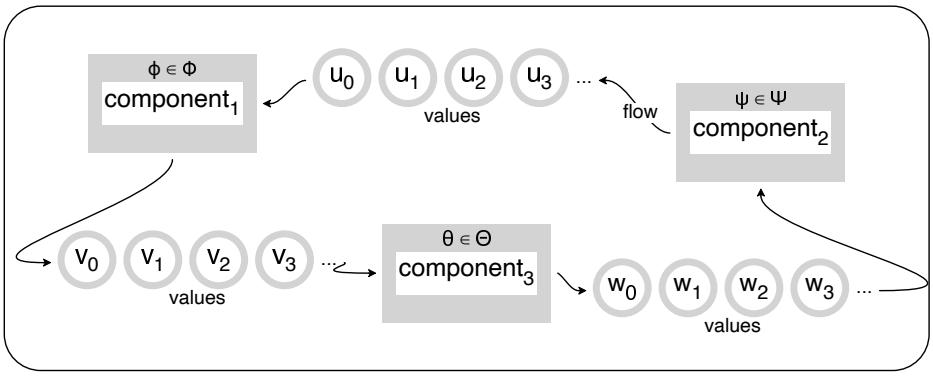


Figure 4 A System with Domain-Specific Contracts

4 Implementing Domain-Specific Contracts

Implementing DSCs demands two ingredients: a low-level contract system and a mechanism for embedding domain-specific syntax. If a language lacks such a mechanism, a developer can use design patterns to construct embedded languages [41].

Figure 4 graphically summarizes how domain-specific contracts function. Each component is combined with a logical contract specification (the gray frame). The component's developers choose the most appropriate logic to use, based on the property to be enforced. These logic languages come with translators that turn these specifications into predicates, which then become pieces of trace contracts. In the diagram, the enforcement mechanism comes into play with the sequences of values that components exchange. The contract code checks the sequence of values that a component receives and sends out. The gray border of these values in Figure 4. These borders indicate that values may acquire a *proxy* layer that contains specification-checking code from the component's interface [17, 50]. A proxy interposes on interactions with the underlying object, e.g., a method call, and performs dynamic checks.

This section starts with a brief tutorial of Racket contracts, focusing on trace contracts (Section 4.1). Next, the section introduces the essentials of Racket's DSL implementation system (Section 4.2) and what features readers should watch out for when implementing DSCs (Section 4.3). Lastly, the section gives some hints concerning the adaptation of this blueprint to languages with different DSL-programming facilities and contract systems (Section 4.4).

4.1 Low-Level Contract Mechanisms

For enforcement, DSCs rely on low-level contract mechanisms for stating and checking Boolean expressions for various kinds of properties: plain pre-conditions and post-conditions for functions; higher-order contracts for imposing invariants on foreign and behavioral values; and trace contracts for checking properties about sequences of calls.

In an untyped programming language, simple software contracts provide the programmer with the means to check type-like properties; in typed languages they supplement method signatures not expressible in the type system. In Racket, contracts are also first-class values: they can be named, composed, and attached to values separately.

Higher-order contracts [17] go beyond plain assertions describing invariants of behavioral values—such as objects—and generating blame information[2] when dynamic[3] checks detect a violation. Here is a vanilla object contract for iterators:

```
(define iterator/c
  (object/c
    [has-next (->m boolean?)]
    [next     (->m any/c)]))
```

This contract specifies the method names along with a contract for the inputs and outputs of each method. When `iterator/c` is attached to an object, it ensures that the object comes with the expected methods and wraps it in a proxy. This wrapper performs checks on the inputs and outputs of the methods according to the stated expectations in method contracts (`->m`). Here, neither method consumes any arguments; both return one value.

An object contract merely enforces the pre-conditions and post-conditions on method arguments and results. It neither states nor checks the requirement of the HasNext property. To do so requires the trace contract extension to Racket's higher-order contract system.

A trace contract [37] collects values that flow through certain code positions and enforces properties of the collected sequence. Hence, trace contracts can check temporal properties such as the HasNext property. Figure 5 shows such a contract.

```
(define iterator/c
  (trace/c ([method* symbol?])
    (object/c
     [has-next (and/c (->m boolean?) (apply/c [method* 'has-next]))]
     [next     (and/c (->m any/c)    (apply/c [method* 'next]))])
    (full (method*) has-next?)))

;; Stream[Symbol] -> Boolean
(define (has-next? s [okay? #false])
  (match s
    [(stream)                  #true]
    [(stream-cons 'has-next s*) (has-next? s* #true)]
    [(stream-cons 'next s*)     (and okay? (has-next? s* #false))]))
```

Figure 5 Checking the HasNext Property

The `iterator/c` contract is constructed with `trace/c`, which consists of three parts:

1. a declaration of a trace variable (`method*`) or several and its element type;
2. a contract (here an `object/c` contract); and
3. a predicate clause (`full`).

The declared trace variable names the stream of collected values. Annotations to the object contract push symbols into this stream whenever the corresponding method is called. Concretely, `apply/c` pushes the symbols `'has-next` and `'next` into `method*` when the

[2] In a higher-order context, values enter contracts as they flow from one component to another. A contract violation may take place in the receiving component or in some other component, unrelated to the two involved in the contract. Hence, tracking the provenance of values is critical so that a contract violation can be related to a contract attachment [17].

[3] On occasion they can be statically verified [38, 53] but this direction is beyond the scope of this paper.

corresponding method is called. Whenever a symbol is added to `method*`, the trace contract runs the predicate from the third part on the full stream. For the iterator contract, the predicate ensures that the trace does not have two consecutive `'next` symbols.

Importantly, each time a trace contract is attached to a value, it initializes a new trace just for that value. That means, different iterators have distinct traces that are separate from one another. State management is completely left to the trace-contract system and not the programmer. Hence, predicates over traces are plain (testable) functions.

Clearly, the direct use of this low-level contract mechanism "drowns" the essence of the iterator contract. By contrast, with a bit of training, every programmer can comprehend the contract in the introduction much more quickly.[4] First, it uses `object-trace/c` which provides a thin layer of syntactic sugar over the repetitive `apply/c` contracts. Second, it uses a one-line regular expression over method names instead of a recursive predicate.

Trace contracts, as originally implemented, were not expressive enough to support `#:include`. The original implementation restricted contracts such that a predicate clause could only depend on traces from the same `trace/c` form. To support `#:include`, the underlying trace-contract system had to be modified to remove this restriction, allowing predicate clauses to depend on an arbitrary trace. By contrast, `#:extend` did not require any changes to the trace-contract system.

4.2 Domain-Specific Languages via Macros

Nearly all modern Lisp-like languages [10, 20, 24, 46, 47], and many languages outside the Lisp family [7, 14, 42], support syntactic abstraction via macros. A Lisp-style macro is a programmer-defined transformation from one syntax tree to another. In its most basic form, a macro is just a simple rewrite rule. But some languages, and Racket in particular, take the idea much further. Racket enables language-oriented programming [15] through macros, so programmers can construct rich DSLs and co-mingle expressions in these DSLs, without ever leaving the host language.

Racket provides one particularly useful tool for constructing DSC implementations: the `syntax-spec` library [2, 3]. Two features of `syntax-spec` make it suitable for creating new DSCs. First, developers can declaratively specify the language grammar, making it straightforward to develop and maintain the Racket-like syntax of DSCs. Second, `syntax-spec` is *binding aware*, meaning it understands the scope of variable declarations. It can thus reliably provide services such as rename refactoring or determining the free variables of a formula.

Figure 6 shows the BNF representation of the PLTL grammar and the `syntax-spec` definition for the same grammar in Racket. These grammars directly correspond to each other, and it is easy to go back and forth. In order to make the `syntax-spec` grammar binding-aware, a developer adds *binding specifications* such as the one for the `exists` form. Here, the binding specification states that in $\exists x.\phi$, the variable x is bound in ϕ.

Implementing the static semantics of a DSC is almost as straightforward as declaring its grammar. Take the monitorability criteria of PLTL [6], which says that a well-formed PLTL formula can be turned into a predicate over a trace if it satisfies the judgment on the left side of Figure 7. The equivalent Racket code is on the right of the figure. Once again, the differences between these two are superficial. The Racket implementation pattern matches on the syntax using the `syntax-parse` system [11]. Each rule in the monitorability

[4] Most programmers are familiar with regular expressions, but other specification languages are less-well understood. For example, LTL is known to pose challenges for developers [22, 23]. Employing such specification languages may necessitate more training on the part of developers.

$$\phi \in \text{Formula} = p \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \exists x.\phi \mid \bullet\,\phi \mid \phi\,\mathsf{S}\,\phi$$

```
(syntax-spec
 (nonterminal pltl-formula
  #:allow-extension pltl-macro
  #:binding-space pltl-space
  p:pltl-pat
  (neg f:pltl-formula)
  (disj2 f1:pltl-formula f2:pltl-formula)
  (conj2 f1:pltl-formula f2:pltl-formula)
  (exists x:pltl-var f:pltl-formula)
  #:binding (scope (bind x) f)
  (previous f:pltl-formula)
  (since f1:pltl-formula f2:pltl-formula)))
```

Figure 6 Grammar of PLTL (Simplified)

judgment corresponds to a branch in the match. In the PLTL pattern case, the function uses the free variables (`fvs`) feature mentioned earlier.[5]

Finally, if a formula satisfies the static semantics, it can be translated into a predicate. Figure 8 shows a simple translation for PLTL. It translates each formula into a tree of structs, which is then interpreted by the PLTL runtime.

4.3 Features To Observe

While the use of the `syntax-spec` library greatly facilitates the declaration and translation of DSCs, this very simplicity obscures a number of aspects that come "for free" but may have to be realized separately in alternative implementations. When implemented with `syntax-spec`, domain-specific contracts signal syntax errors at the appropriate level, have direct support from the IDE, and are syntactically extensible.

Static Errors

A `syntax-spec` DSC implementation can automatically synthesize reasonable syntax errors for the new notation. For example, ill-formed PLTL formulas are identified like this:

```
> (define-pltl φ
    (S 'next))
expected more terms in:  (S 'next)
```

In ϕ, the S operator expects another operand, so the error points out an arity mismatch.

Similarly, a `syntax-spec` implementation can issue an informative message when static checking fails. As mentioned in the preceding subsection, a syntactically well-formed PLTL formula may be *invalid* because it cannot be translated into a predicate (as is):

[5] A straightforward feature to add to this static semantics is a monitorability search procedure. Some tools, such as MonPoly [6], attempt to rewrite nonmonitorable formulas into monitorable ones if possible. The same can easily be done with DSCs, immediately after the monitorability check fails. Since macros are implemented using regular Racket code, such a procedure can make use of any heuristic search libraries available in the host.

$$\frac{\rho \vdash \phi \qquad \rho \vdash \psi}{\rho \vdash \phi \vee \psi}$$

$$\frac{\rho_\phi \vdash \phi \qquad \rho_\psi \vdash \psi \qquad \rho_\phi \subseteq \rho_\psi}{\rho_\phi \cup \rho_\psi \vdash \neg\phi \wedge \psi}$$

$$\frac{\rho_\phi \vdash \phi \qquad \rho_\psi \vdash \psi \qquad \rho_\phi \subseteq \rho_\psi}{\rho_\phi \cup \rho_\psi \vdash \neg\phi \,\mathsf{S}\, \psi}$$

$$\frac{\rho_\phi \vdash \phi \qquad \rho_\psi \vdash \psi \qquad \rho_\phi \subseteq \rho_\psi}{\rho_\phi \cup \rho_\psi \vdash \phi \,\mathsf{S}\, \psi}$$

$$\frac{\rho_\phi \vdash \phi \qquad \rho_\psi \vdash \psi}{\rho_\phi \cup \rho_\psi \vdash \phi \wedge \psi} \qquad \frac{\rho \cup \{x\} \vdash \phi}{\rho \vdash \exists x.\phi}$$

$$\frac{\rho \vdash \phi}{\rho \vdash \bullet\phi} \qquad \frac{\rho = \mathrm{FV}(p)}{\rho \vdash p}$$

```
(define (monitorable? stx)
  (syntax-parse stx
    [(disj2 f g)
     (and (equal? (fvs #'f) (fvs #'g))
          (monitorable? #'f)
          (monitorable? #'g))]
    [(conj2 (neg f) g)
     (and (subset? (fvs #'f) (fvs #'g))
          (monitorable? #'f)
          (monitorable? #'g))]
    ── similar cases ──
    [(conj2 f g)
     (and (monitorable? #'f)
          (monitorable? #'g))]
    [(exists (x:id ...) f)
     (monitorable? #'f)]
    ── similar cases ──
    [(neg _) #false]
    [p #true]))
```

Figure 7 Static Semantics for PLTL (Simplified)

```
(define-syntax (translate stx)
  (syntax-parse stx
    [(_ (neg f))              #'(neg-s (translate f))]
    [(_ (disj2 f g))          #'(disj2-s (translate f) (translate g))]
    ── similar cases ──
    [(_ (exists (x ...) f)) #'(exists-s (set 'x ...) (translate f))]
    [(_ x:id)                 #'x]
    [(_ p)                    #'(translate-pattern p)]))
```

Figure 8 Translator for PLTL (Simplified)

```
> (define-pltl ψ
    (∃ (t) (¬ (• t))))
not monitorable in:  (¬ (• t))
```

Even if a PLTL formula is well-formed and valid, it can still be misused:

```
> (+ ψ 13)
cannot use PLTL formula here:  ψ
```

The `define-pltl` form creates definitions in a separate *binding space* as required by the one-line `#:binding-space` option in Figure 6. Languages, including domain-specific notations, are often composed of several syntactic categories that are mixed at particular interface points. For example, Java has a syntactic category of expressions and types that are interleaved in a certain way; definitions created in one do not exist in the other. The `syntax-spec` implementation of PLTL creates its own binding space, distinct from the one for expressions, meaning that a reference to a PLTL name from an expression context signals a static error.

IDE Integration

The `syntax-spec` library facilitates the integration of a DSC implementation with Racket IDEs. One such IDE is DrRacket [16]; another one is Racket Mode for Emacs.[6]

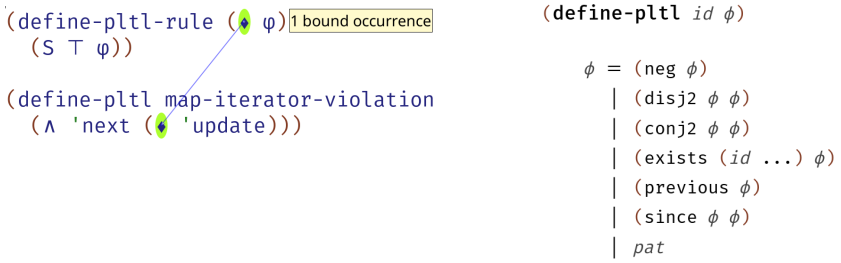


Figure 9 IDE Services for PLTL

Figure 9 shows two such services. First, when a user hovers over an identifier, arrows are drawn from the binding occurrence of an identifier to every reference (Figure 9, left). Here, the definition of ♦ points to its use in the formula. The IDE has a rich understanding of the binding structure of the DSC, which can be used to support reliable program transformations including rename refactorings. Second, if a DSC comes equipped with documentation, as PLTL does, this documentation can be displayed directly in the IDE. When the cursor is placed inside an identifier with recognized documentation, a box containing a summary of the documentation appears in the upper right corner of the IDE. For `define-pltl`, that summary is the grammar of primitive PLTL forms (Figure 9, right). This service is available because Racket's documentation language Scribble [19] is binding aware.

User Extensibility

PLTL comes with just two primitive temporal operators. Although derived operators add no expressive power to the logic, they are necessary for writing concise specifications. While the PLTL library could provide direct support for such derived operators, doing so has the serious downside of complicating the static semantics, the translator, and the runtime system, which would have to deal with each kind of operator separately.

Instead, a `syntax-spec` implementation of a DSC can be made syntactically extensible with a single annotation: `#:allow-extension`. This `#:allow-extension` option opens up the `pltl-formula` nonterminal for simple macro extensibility. Using this capability, even a user can define additional operators in straightforward manner:

```
(define-pltl-rule ⊤
  (? (λ _ #true)))

(define-pltl-rule (♦ ϕ)
  (S ⊤ ϕ))
```

The `syntax-spec`-generated `define-pltl-rule` form defines a syntactic transformation that rewrites all instances of ⊤ and ♦ϕ in a PLTL formula: ⊤ is rewritten to a predicate that always returns `#true`, meaning a machine that succeeds on all streams of collected values; and ♦ is rewritten to a formula that requires nothing to be true until ϕ holds.

[6] Maintained by Greg Hendershott; see `https://racket-mode.com`.

These syntactic transformations are seamlessly integrated into PLTL. For example, a user can write a utility library with these abbreviations and export the PLTL library flexible to use and simplify its implementation tremendously.

Compositionality

For the illustrative example of this paper, it is also possible for users to compose contracts that may have been written independently into a single interface contract. For example, the iterator contract from Figure 2 can easily be augmented to additionally check the HasNext property, even though these specifications may have been defined separately:

```
(define (map-iterator*/c map/c)
  (and/c has-next/c (map-iterator/c map/c)))
```

The `and/c` form comes from Racket's contract library and enforces that values satisfy both conjuncts (in order). It may compose *any* kind of contract that compiles to the already-mentioned proxy mechanism in Racket.

Blame Assignments

In the case of Racket and the DSCs in this paper, contracts inherit the blame-assignment mechanism from Racket [17]. That is, once a dynamic contract check discovers a violation, it raises an exception that assigns blame to a particular component. Here is an excerpt of the error message that explains a violation of the HasNext property:

```
> (define m (new map% ——))
> (define c (send m keys))
> (define t (send c iterator))
> (send t next)
next: contract violation
blaming: top level
```

An iterator created from a collection has its `next` method called without a corresponding call to `has-next`. The contract system issues a blame assignment that points to the method that failed and the component that violated the specification—here, the top-level code. This information might be useful for tracking down the source of bugs in large systems [28, 29, 30].

4.4 Ingredients for Adding DSCs to a Programming Language

The DSC architecture is generic and should apply to many programming languages, with varying degrees of ease. It requires two essential ingredients: (1) facilities for creating embedded DSLs for the desired logics, such as macros; (2) a mechanism for dynamically checking logical formulas, such as contracts. The quality of these ingredients in a language determines how easy it is to instantiate this architecture and how many of the features from the preceding subsection can be realized. For example, a programming language with powerful metaprogramming facilities reduces the notational overhead on programmers; without such facilities, a DSC implementer may have to resort to shallow or deep embedding techniques [21], which may impose extra burden on developers.

Metaprogramming

A metaprogramming system enables programmers to extend their language with new linguistic constructs and collect them in libraries or frameworks. Ideally, it permits blurring the

distinction between built-in constructs and user-defined ones. Importantly, metaprogramming facilities exist *inside* the programming language. Depending on the chosen language, metaprogramming facilities can be used in the small and large: from quick syntactic abbreviations all the way up to full feature-rich special-purpose DSLs. For DSCs, the metaprogramming facilities must be powerful enough to create the *embedded* DSLs necessary for expressing logical specifications.

Different languages have vastly different metaprogramming mechanisms. Some languages provide lightweight syntactic metadata: decorators in Python and annotations in Java. In other languages such as Ruby, the syntax is sufficiently flexible and the semantics sufficiently dynamic, that a DSC notation can be built without any explicit language support. Another major class of metaprogramming facilities are those based on (syntax-tree) *macros*: compile-time rewriting rules. Before code generation, the compiler applies these rules to obtain the program in a core syntax, a process often called *macro expansion*. Languages in the Lisp family are especially well-known for such macros, such as Clojure [24], but others such as Rust and Scala have adopted them too. Parenthetical syntax is not necessary [18].

Racket is well-suited for defining DSCs as it provides a great deal of support for language-oriented programming [15]. While its advanced macro system essentially makes up a nearly complete compiler API, the implementation of a DSC needs few of these capabilities.

Dynamic Checking

A contract system consists of a domain-specific notation and a run-time mechanism for dynamically checking a program's behavior. The notation has to provide the means to identify relevant events, e.g., "the first argument to the method call `o.m`", and the predicate that should hold of such events, e.g., "is a positive integer." These statements are usually understandable as ordinary program code—Boolean-valued assertions—and are easily translated into run-time checks. The runtime system issues a descriptive error when any of these checks fail.

Modern contract systems permit programmers to monitor specifications over any kind of value, including higher-order values such as objects, functions, and even the kind of first-class classes found in Python, Racket, and Ruby. The corresponding run-time checks rely on proxy values that wrap values flowing through contracts and contain the Boolean-valued run-time checks. Direct support for proxy values is available in a number of languages, e.g., JavaScript [51]. Even if proxies are unavailable, implementing proxies in some dynamic languages does not require explicit support. Ruby, for example, supports objects that can interpose on all `send` method invocations. Such a facility has been used to implemented higher-order contracts [49]. So, while contracts are implemented as a core feature in some languages, such as Eiffel [33, 34], they can be implemented as a library to some extent, especially if proxies are available.

Racket comes with a pioneering higher-order contract system [17]. The contract notation is realized as an embedding, via a rich set of contract combinators; the base contracts are predicates as in any other system. With trace contracts [37], programmers can articulate predicates over traces of values that flow through particular points in a contract. Trace contracts make it particularly straightforward to bridge the gap between the DSC specifications presented here and the low-level contract system itself.

5 Evaluating the Domain-Specific Contracts Approach

An assessment of the DSC idea and the presented instance of DSCs for trace contracts must address several aspects: (1) whether DSCs can accommodate existing logic notations;

(2) what developer-facing features these implementations provide; (3) and how precisely DSCs can report specification violations. The first two points call for identifying and re-implementing a corpus of logic notations for specifications in the literature (Section 5.1) and for an assessment of the qualities of their implementations (Section 5.2). The last point is a matter of implementation quality; an extensive test suite can validate that the blame assignment points to the problematic party for some common examples of properties from the literature (Section 5.3). In addition, this section relates the DSC idea to prior work.

5.1 Corpus of Logics

The literature proposes a wide variety of logic notations for specifications that are suitable to check with trace contracts. Here are six common ones:

Past-Time Linear Temporal Logic (PLTL) as covered in the preceding sections.

Regular Expressions (RE) are a simple formalism equivalent in expressive power to finite state machines [27]; they are a well-known way for specifying parts of the behavior of a software system [45].

Finite State Machines (NFA, DFA) have a finite number of states and transitions between them. A deterministic finite-state machine (DFA) denotes the sequences of events that, when run on the state machine, follow a unique path that ends in an accepting state [43]. A non-deterministic finite-state machine (NFA) denotes the sequences of events that, when run on the state machine, end in an accepting state along *some* path.

Quantified Event Automata (QEA) resemble DFAs but are augmented with extra features [4]. As the name suggests, the machine supports quantification. Conceptually, a QEA represents a family of automata—one for every possible instantiation of quantified variables. Practical implementations of QEA more efficiently compute transitions via a compact representation of this family of automata.

Stream Logic (SL) describes properties by defining stream equations [12]. A stream is a sequence of events that is accessible with a finite lookback. The dynamic checks for SL specifications compute the value for each stream at the current index, making use of the static dependencies of equations to solve them in the correct order. Irrelevant values, i.e., ones that are beyond the finite lookback, are automatically discarded.

The authors have adapted all these to the DSC architecture, and the following two subsections assess this re-implementation effort.

5.2 Qualitative Assessment

The implementations of these six specification logic in the DSC architecture have distinct qualities from the perspective of language creators and users. This subsection reports those qualities in terms of four dimensions: binding positions, definitions, static semantics, and macro extensibility. Figure 10 summarizes the discussion in a table.

	PLTL	RE	DFA	NFA	QEA	SL
Binding Rules	✓	×	×	×	✓	✓
Definitions	✓	✓	×	×	×	×
Static Semantics	✓	×	×	×	×	✓
Macro Extensible	✓	✓	×	×	×	×

Figure 10 Summary Assessment of Qualitative of the Logic DSLs

Binding Positions

Some DSLs are variable-free; others come with variable declarations that introduce binding positions and determine the scope of a declaration. Since the `syntax-spec` system comes with a binding-rules mechanism, it allows DSC creators to specify which pieces of a formula are variable references, which are binding positions, and the scope of bindings. By declaring the binding structure, the presented DSC implementations can automatically support IDE services, static-semantics passes, and translations into predicates.

Logic notations without quantification, i.e., RE, DFA, and NFA, do not require binding rules. Specifically, the state-machine notations do not use up-front declaration of states; if they did, binding rules would be required.

By contrast, both PLTL and QEA come with variable quantification, and quantified variables are binders. In SL, equations must reference other equations or primitive event streams. The binding rules for SL guarantee that there are no free variables referencing an undefined stream.

Separate Definitions

For some logic notations, formulas can be defined piecemeal via the binding space mechanism. Developers can build up libraries of re-usable formulas that others can then compose into specifications of modules, classes, and other components. For such compositional notations, definition forms are a highly useful mechanism. While the state-machine notations from the literature do not seem compositional, both RE and PLTL greatly benefit from the introduction of definition forms.

Static Semantics

Some specification notations impose criteria that determine whether well-formed formulas are valid. If an implementation can enforce such static-semantics constraints, it is superior to plain frameworks or functional libraries, because it informs developers at compile time about basic mistakes instead of delaying the discovery of specification mistakes until run time. Programmers can thus avoid unexpected run-time errors and dynamic debugging due to invalid specifications.

While Racket's `syntax-spec` library automatically guarantees that logical specifications are well-formed and closed with respect to any binding rules, the static semantics of DSCs must be written as an additional check. Of the investigated notations, PLTL and SL must satisfy such criteria. Section 4 covers the monitorability criteria for PLTL. For SL, the DSL implementation statically checks that the dependency graph of the stream equations has no closed walk with a total weight of 0, where the weights are determined by indices into streams. This check statically generates the dependency graph from the stream equations via local macro expansion, even when references to streams are embedded inside Racket sub-expressions.

Macro Extensibility

When logical formulas are compositional, programmers benefit from a macro-extensible language. The RE and PLTL notations make heavy use of this feature because many useful operators can be derived from a small set of primitive operators. By expanding the surface syntax to a small core language, the implementation can be organized like an ordinary compiler (e.g., for functional languages such as Haskell, ML, or Scheme). Equally important,

macro extensibility enables users to create syntactic abstractions when plain functional abstractions do not suffice to hide repeated patterns in formulas.

In summary, the re-implementation of these specification notations preserve all desirable aspects (e.g., the static checks) and, indeed, go beyond usual implementations, offering users additional affordances. Furthermore, DSCs are embedded into the host, meaning the host's IDEs (DrRacket, Emacs) support DSCs essentially as well as plain Racket programs. When implemented correctly, users do not notice any friction.

Property	Logic / SLOC	Description
MapIterator	PLTL (7), RE (1)	An `iterator` constructed from a collection, which is itself constructed from a `map`, is invalidated once the underlying `map` is mutated.
HasNext	QEA (6), RE (1)	A call to the `has-next` method must precede every call to the `next` method.
HashCode	QEA (5), SL (7)	An object's `hash-code` must not change so long as it is a being used as a key in a map.
ClosedReader	SL (8)	A call to the `read` method of a `reader` object must not take place after the underlying input stream has been closed.
SetFromMap	SL (6)	Creating a `set` from a `map` renders the latter unusable.
RemoveOnce	RE (1)	An `iterator`'s `remove` method can be called only once per invocation of the `next` method.
StreamClose	RE (1)	A `stream` may be used until it is closed or the underlying TCP listener is closed.
PortExclusivity	QEA (5)	Listeners can be created for a port, so long as there is no active listener already associated with that port.

Figure 11 Sample Properties

5.3 Blame Assignment

One claimed benefit of contracts is that error messages come with blame information identifying the source of the specification violation. Blame assignment in higher-order settings, and even more so for temporal properties, is subtle. In general, blame assignment points to the components that contribute to a violation of a specification.

To understand whether blame assignment from the underlying contract system works well for DSCs, it is necessary to adapt examples from the literature and to test violations on sample code snippets. That is, unit tests should validate that the Racket implementation actually produces blame assignments in accordance with the source of the bug.

Figure 11 list some sample properties from the literature, which logics they were implemented in, and how many lines it took to write each formula. All of them explain properties of the Java API that are stated informally but not checked via the assertion system. Other systems can enforce these properties [31], and so can the Racket implementation of DSCs.

The test suite for the Racket implementation contains adaptations for all of these properties. The corresponding code pieces come with components that violate the properties. Unit tests enforce that the checking of these DSC specifications catches all of these violations and that the relevant blame assignments point to the violating component.

5.4 Prior Work

Dimoulas et al. [13] present the first and only analysis in the contract-systems literature that explicitly mentions the need for a contract-specific notation. Their analysis explains a contract system as resting on three pillars: (1) a notation for expressing contracts; based on (2) a set of interposition points in the run-time system that allows the inspection of values; and (3) a syntactic mechanism for attaching contracts to values. Generally speaking, research on contracts has tightly coupled the three, associating one contract notation with specific interposition points and attachment mechanism. Instead, the DSC architecture proposes that any language with contracts benefits from a *loose coupling* between contract notations on the one hand, and attachment mechanisms and interposition points on the other.

JavaMOP [8], a run-time verification system, is the only existing system with a similarly loose coupling. From a high-level perspective, JavaMOP rests on an aspect-like interposition system, dubbed monitor-oriented programming (MOP) [32]. Since JavaMOP is a run-time verifier, programmers do not formulate method-level or even class-level contracts, but whole-program properties. The JavaMOP framework compiles these properties into (a) monitoring code snippets that are woven into the program at appropriate join points and that send values to (b) a parallel property-checking process.

Unlike other run-time verification systems, JavaMOP permits programmers to choose the property notation best suited to the task and, indeed, allows them to create and plug in their own notation. From this angle, JavaMOP resembles the Racket implementation of the proposed DSC architecture.

However, three differences stand out. First, the DSC proposal is intended to be realizable in any programming language. As detailed in Section 4, an implementation can be realized with minimal effort in languages with macro systems such as Racket, Clojure, Julia, Rust, or Scala. In languages without macro systems, the realization requires a bit more labor, depending on existing features, but using conventional techniques for embedding domain-specific languages make DSCs feasible, even without significant engineering resources. Second, the end-product of DSCs is more convenient for programmers to use than a toolchain-based solution such as JavaMOP. A library can be installed directly from a language's package manager and used immediately. Finally, DSCs provide additional benefits beyond what JavaMOP delivers, e.g., IDE integration and blame assignment (from the underlying contract system, if available). The remainder of this section provides a detailed comparison.

Logic Plugins

JavaMOP users can develop their own logic plugins to add new, possibly domain-specific specification languages. These plugins are essentially ordinary compilers from the surface syntax to Java code. Constructing such a compiler requires the use of traditional compiler generator tools, such as a parser generator, and a solid understanding of the semantics and behavior of the target language. As a result, developing a new logic or extending an existing one is challenging and not a lightweight exercise.

In the context of the proposed DSC architecture, the development of DSCs relies on the host language's existing metaprogramming facilities or established DSL implementation patterns. In a language with a macro system, such as Racket, new DSCs with sophisticated features can be created easily with a combination of declarative grammar specifications and recursive functions. And, if the host allows for declarative DSL specifications, these DSCs are maintainable in a straightforward manner.

Linguistic Integration

A byproduct of JavaMOP's extra-linguistic approach is that logical specifications are syntactically isolated from the rest of the program. JavaMOP specifications are typically added as comments to the code or as entirely separate text files. As a consequence, these artifacts end up as second-class citizens within a software project.

By contrast, DSCs make logical specifications an integrated part of the underlying project; they exist *within* the program. As such, all host-language tools continue to work with logical specifications. Notably, DrRacket's existing IDE services seamlessly extend to DSC code.

Trace Management

JavaMOP takes a global perspective on monitoring. System events are captured and stored in a global sequence of events. The global monitoring strategy is central to the run-time verification approach. Since checking globally generated traces requires a slicing technique, JavaMOP's implementation makes slicing independent of the underlying logic to reduce the burden on the developer of a new logic notation [8, 25]. In the end, however, the program-global property approach remains an obstacle to reasoning about code in a compositional manner. Slicing also requires a notion of equality on objects which, if there is not one defined already, relies on pointer equality. Depending on pointer equality for objects violates a key principle of object-oriented programming [9].

Contracts, by contrast, often eliminate the need for slicing because the state for a given value is freshly initialized when the value passes through an attachment boundary. Monitoring is local. Thus, the MapIterator property from Section 2 can be attached to a specific object and remains quantifier free; the equivalent formula in JavaMOP requires quantifiers and hence a slicing implementation of the monitoring process. Localized monitoring also unlocks unique advantages such as blame assignment (if the underlying contract system provides it). Slicing is still occasionally useful in contracts, for example, Section 3 shows slices based on port numbers. A notion of equality is defined for port numbers, so the principles of good object-oriented design are maintained.

Run-time Enforcement

As originally published, JavaMOP supported only one run-time enforcement mechanism. The AspectJ [26] weaver would statically inject monitoring code into a system based on the given specification. This requires AspectJ as a compile-time dependency and alters the ordinary build process for a Java project. Now, JavaMOP supports a dynamic enforcement mechanism known as the JavaMOP agent. With this mode, an instrumented JVM supplies the means to record system events during execution. This offers some advantages over the static approach, but requires a specialized runtime. Both of these mechanisms are somewhat unique to Java and are not easily available in other languages.

Contracts use a completely different implementation approach. While first-order contracts essentially get away with Boolean expressions, higher-order and trace contract systems rely on just proxy objects to monitor a running program. If the language does not support proxy objects directly, implementing a reasonable approximation is feasible. The key is that proxies enable a simple, local-enforcement mechanism, which is important for blame assignment in higher-order settings.

6 A Future for Domain-Specific Contracts

Racket makes it straightforward to instantiate the DSC idea. It comes with an expressive library of contract combinators and a powerful facility for creating domain-specific notations. Leveraging the latter allows the creators of specific DSCs to implement notations with little effort and translate them to contract expressions. End-users can employ a library that provides DSCs just like any other dependency—critically, while continuing to use, and benefiting from, existing IDEs.

While the Racket implementation is particularly straightforward, several programming languages offer similar combinations of metaprogramming and contract features. For example, Clojure has adapted most of Racket's contract system and comes with its own macro system. Scala too has a rich metaprogramming system for embedding DSLs (e.g., [44]) and could be equipped with a contract system based on Java's assertion system.

This paper has discussed the DSC idea only in the context of trace contracts with temporal specification languages. However, the idea of combining contracts with domain-specific specification languages goes far beyond that particular instantiation. For example, effect-handler contracts [36] combined with an authorization logic [1] yields authorization contracts [35] that can enforce security properties. Or, network-aware contracts [52] combined with a DSL for describing the legal moves of a game [13] yields the rule checker for an implementation of an online game. Indeed, there are plenty of other combinations, not yet studied and worthy of exploration from the DSC perspective. Contracts and DSLs are useful alone. But together, they are better.

References

1 Owen Arden, Jed Liu, and Andrew C. Myers. Flow-Limited Authorization. In *Computer Security Foundations (CSF)*, 2015. doi:10.1109/csf.2015.42.

2 Michael Ballantyne and Matthias Felleisen. Injecting Language Workbench Technology into Mainstream Languages. In *Eelco Visser Commemorative Symposium (EVCS)*, 2023. doi:10.4230/OASIcs.EVCS.2023.3.

3 Michael Ballantyne, Mitch Gamburg, and Jason Hemann. Compiled, Extensible, Multi-Language DSLs (Functional Pearl). In *International Conference on Functional Programming (ICFP)*, 2024. doi:10.1145/3674627.

4 H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard. Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In *Formal Methods (FM)*, 2012. doi:10.1007/978-3-642-32759-9_9.

5 David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. Monitoring Metric First-Order Temporal Properties. *Journal of the ACM*, 2015. doi:10.1145/2699444.

6 David Basin, Felix Klaedtke, and Eugen Zălinescu. The MonPoly Monitoring Tool. In *Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*, 2017. doi:10.29007/89hs.

7 Eugene Burmako. Scala Macros: Let Our Powers Combine! In *Workshop on Scala*, 2013. doi:10.1145/2489837.2489840.

8 Feng Chen and Grigore Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2007. doi:10.1145/1297027.1297069.

9 William R. Cook. On understanding data abstraction, revisited. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2009. doi:10.1145/1640089.1640133.

10 Ryan Culpepper. Fortifying Macros. *Journal of Functional Programming (JFP)*, 2012. doi:10.1017/s0956796812000275.

11 Ryan Culpepper and Matthias Felleisen. Fortifying Macros. In *International Conference on Functional Programming (ICFP)*, 2010. doi:10.1145/1863543.1863577.

12 B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H.B. Sipma, S. Mehrotra, and Z. Manna. LOLA: Runtime Monitoring of Synchronous Systems. In *Temporal Representation and Reasoning (TIME)*, 2005. doi:10.1109/TIME.2005.26.

13 Christos Dimoulas, Max S. New, Robert Bruce Findler, and Matthias Felleisen. Oh Lord, Please Don't Let Contracts Be Misunderstood (Functional Pearl). In *International Conference on Functional Programming (ICFP)*, 2016. doi:10.1145/2951913.2951930.

14 Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-Based Syntactic Language Extensibility. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2011. doi:10.1145/2048066.2048099.

15 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A Programmable Programming Language. *Communications of the ACM (CACM)*, 2018. doi:10.1145/3127323.

16 Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming (JFP)*, 2002. doi:10.1017/S0956796801004208.

17 Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *International Conference on Functional Programming (ICFP)*, 2002. doi:10.1145/581478.581484.

18 Matthew Flatt, Taylor Allred, Nia Angle, Stephen De Gabrielle, Robert Bruce Findler, Jack Firth, Kiran Gopinathan, Ben Greenman, Siddhartha Kasivajhula, Alex Knauth, Jay McCarthy, Sam Phillips, Sorawee Porncharoenwase, Jens Axel Søgaard, and Sam Tobin-Hochstadt. Rhombus: A New Spin on Macros without All the Parentheses. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2023. doi:10.1145/3622818.

19 Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *International Conference on Functional Programming (ICFP)*, 2009. doi:10.1145/1596550.1596569.

20 Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. https://racket-lang.org/tr1/.

21 Jeremy Gibbons and Nicolas Wu. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). In *International Conference on Functional Programming (ICFP)*, 2014. doi:10.1145/2628136.2628138.

22 Ben Greenman, Siddhartha Prasad, Antonio Di Stasio, Shufang Zhu, Giuseppe De Giacomo, Shriram Krishnamurthi, Marco Montali, Tim Nelson, and Milda Zizyte. Misconceptions in finite-trace and infinite-trace linear temporal logic. In *Formal Methods*, 2024. doi:10.1007/978-3-031-71162-6_30.

23 Ben Greenman, Sam Saarinen, Tim Nelson, and Shriram Krishnamurthi. Little tricky logic: Misconceptions in the understanding of LTL. *The Art, Science, and Engineering of Programming*, 2023. doi:10.22152/programming-journal.org/2023/7/7.

24 Rich Hickey. A History of Clojure. In *History of Programming Languages (HOPL)*, 2020. doi:10.1145/3386321.

25 Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Rosu. Garbage Collection for Monitoring Parametric Properties. In *Programming Language Design and Implementation (PLDI)*, 2011. doi:10.1145/1993498.1993547.

26 Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001. doi:10.1007/3-540-45337-7_18.

27 S.C. Kleene. Representation of Events in Nerve Nets and Finite Automata. In *Automata Studies*. Princeton University Press, Princeton, N.J., 1956. doi:10.1515/9781400882618-002.

28 Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. How to evaluate blame for gradual types. In *International Conference on Functional Programming (ICFP)*, 2021. doi:10.1145/3473573.

29 Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. How to evaluate blame for gradual types, part 2. In *International Conference on Functional Programming (ICFP)*, 2023. doi:10.1145/3607836.

30 Lukas Lazarek, Alexis King, Samanvitha Sundar, Robert Bruce Findler, and Christos Dimoulas. Does Blame Shifting Work? In *Principles of Programming Languages (POPL)*, 2020. doi:10.1145/3371133.

31 Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. How Good are the Specs? A Study of the Bug-Finding Effectiveness of Existing Java API Specifications. In *Automated Software Engineering (ASE)*, 2016. doi:10.1145/2970276.2970356.

32 Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An Overview of the MOP Runtime Verification Framework. *International Journal on Software Tools for Technology Transfer*, 2011. doi:10.1007/s10009-011-0198-6.

33 Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

34 Bertrand Meyer. Applying "Design by Contract". *Computer*, 1992. doi:10.1109/2.161279.

35 Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. Extensible Access Control with Authorization Contracts. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2016. doi:10.1145/2983990.2984021.

36 Cameron Moy, Christos Dimoulas, and Matthias Felleisen. Effectful Software Contracts. In *Principles of Programming Languages (POPL)*, 2024. doi:10.1145/3632930.

37 Cameron Moy and Matthias Felleisen. Trace Contracts. *Journal of Functional Programming (JFP)*, 2023. doi:10.1017/S0956796823000096.

38 Phúc C. Nguyễn, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Soft Contract Verification for Higher-Order Stateful Programs. In *Principles of Programming Languages (POPL)*, 2018. doi:10.1145/3158139.

39 Oracle. Iterator(Java SE 23 & JDK 23). https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Iterator.html, 2024.

40 Oracle. Map (Java SE 23 & JDK 23). https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Map.html, 2024.

41 Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 2009.

42 The Rust Project. The Rust Reference: Procedural Macros. https://doc.rust-lang.org/reference/procedural-macros.html, 2024.

43 M. O. Rabin and D. Scott. Finite Automata and their Decision Problems. *IBM Journal of Research and Development*, 1959. doi:10.1147/rd.32.0114.

44 Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Generative Programming and Component Engineering (GPCE)*, 2010. doi:10.1145/1868294.1868314.

45 Usa Sammapun and Oleg Sokolsky. Regular Expressions for Run-Time Verification. In *Automated Technology for Verification and Analysis (ATVA)*, 2003.

46 Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised^6 Report on the Algorithmic Language Scheme. *Journal of Functional Programming*, 2009. doi:10.1017/s0956796809990074.

47 Guy L. Steele. *Common Lisp the Language*. Digital Press, 1990.

48 T. Stephen Strickland, Christos Dimoulas, Asumu Takikawa, and Matthias Felleisen. Contracts for first-class classes. *Transactions on Programming Languages and Systems (TOPLAS)*, 2013. doi:10.1145/2518189.

49 T. Stephen Strickland, Brianna M. Ren, and Jeffrey S. Foster. Contracts For Domain-Specific Languages In Ruby. In *Dynamic Languages Symposium (DLS)*, 2014. doi:10.1145/2661088.2661092.

50 T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and Impersonators: Run-Time Support for Reasonable Interposition. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2012. doi:10.1145/2384616.2384685.

51 Tom Van Cutsem and Mark S. Miller. Proxies: Design Principles for Robust Object-oriented Intercession APIs. In *Dynamic Languages Symposium (DLS)*, 2010. doi:10.1145/1869631.1869638.

52 Lucas Waye, Stephen Chong, and Christos Dimoulas. Whip: Higher-Order Contracts for Modern Services. In *International Conference on Functional Programming (ICFP)*, 2017. doi:10.1145/3110280.

53 Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static Contract Checking for Haskell. In *Principles of Programming Languages (POPL)*, 2009. doi:10.1145/1480881.1480889.