

Contracts for Design

Improved Error-Messaging in FSM

Shamil Dzhatdoyev¹ and Josephine A. Des Rosiers² and Joshua M. Schappel³
and Marco T. Morazán⁴

¹ Axoni, USA shamild11@gmail.com

² IBM Global Business Services Inc., USA

josiadesrosiers@gmail.com

³ Futuri Media, USA

jmschappel12@gmail.com

⁴ Seton Hall University, USA morazanm@shu.edu

Abstract. Contracts are commonly used to improve software reliability. They effectively assign the blame of an error to the code that commits the error and not to the code where the error is manifested. This makes them well-suited to implement tailored-made error-messaging systems. This article describes a new contract-based error-messaging system for **FSM**—a domain specific language embedded in **Racket** for the automata theory classroom. This system makes extensive use of contracts to assign blame without prescribing solutions and to generate narrowly focused and short error messages. The described error-messaging system has two novel features. The first is that it provides support for the program by design methodology. That is, the error-messaging system is tightly coupled with the steps of a design recipe for state machines. Each error indicates the design-recipe step that has not been successfully completed. The second is that testing is integrated into the error-messaging system. That is, for machines that decide or semidecide a language the programmer may provide words that ought to be accepted and words that ought to be rejected. If a given word produces the wrong result then an error message is generated and machine construction fails.

1 Introduction

Contracts have been extensively used to build (more) reliable software. That is, contracts are used to build software that is correct and robust [7,8]. The use of contracts leads to three desirable features in software systems. First, they help tame the size of software units. This is achieved by *contracting* out subtasks to separate the concerns into different software units. Second, it facilitates correct software reuse. Several subtasks may need to perform the same operation (on different data) and such an operation ought to be contracted out. A contract helps guarantee that the caller provides appropriate arguments and that the callee returns an appropriate value. Third, contracts that assign blame for errors are more effective in helping programmers debug than simply using exceptions

[4,13]. In essence, contracts place assertions on the input and the output known as preconditions and postconditions. The caller is obliged to meet the precondition for each argument and the callee is obliged to meet the postcondition.

An important contract characteristic is the assignment of blame. Commonly, error-messaging systems throw an error at the point where an error is manifested (i.e., caught) and not at the point where the error actually occurs. Consider, for example, the following Racket function to add the numbers in a list of numbers:

```
(define (addlist lst)
  (foldl (λ (n acc) (+ n acc)) 0 lst))
```

Misusing the function, `(addlist '(1 2 c))`, yields an error message where the error is manifested:

```
+: contract violation
expected: number?
given: 'c
```

This error message suggests that the error occurs with the input to `+`. This, however, is rather inaccurate and of little help to the programmer. The real blame for the error lies with the argument provided to `addlist`, which expects a list of numbers as input. To obtain a more robust error message, `addlist` is defined with a contract as follows:

```
(define/contract (addlist lst)
  (-> (listof number?) number?)
  (foldl (λ (n acc) (+ n acc)) 0 lst))
```

The contract combinator, `->`, states that the input (i.e., `lst`) must satisfy the contract `(listof number?)` and the returned value must satisfy the contract `number?`. These types of contracts are known as *flat contracts*—they are fully checked immediately for a given value and are, in essence, predicates [3, Chapter 8]. Evaluating `(addlist '(1 2 c))` yields:

```
addlist: contract violation
expected: number?
given: 'c
in: an element of
    the 1st argument of
    (-> (listof number?) number?)
contract from: (function addlist)
blaming: C:\Users\...
```

Although verbose, the error message now correctly blames the error on the argument provided to `addlist`. The expected input, `lst`, is not a list of numbers because it contains the symbol `c`.

The verbosity of the error message may be trimmed by defining a flat contract with a *projection function* for the behavior of applying the contract. A projection function is a curried function that consumes two arguments: a blame object and

a value to protect. This function may format the error message. It tests the value to protect. If an error is not encountered the value is returned. Otherwise, a blame error is raised using the formatted error message as follows:

```
(define is-lon?
  (make-flat-contract
    #:projection
    (lambda (blame)
      (lambda (x)
        (current-blame-format
          (λ (blame x msg)
            (format "~a: expected a list of numbers but got ~a" msg x)))
        (if (andmap number? x)
            x
            (raise-blame-error blame x "addlist"))))))))

(define/contract (addlist lst)
  (-> is-lon? number?)
  (foldl (lambda (n acc) (+ n acc)) 0 lst))
```

The contract for `add-list` combines two flat contracts: `is-lon?` and `number?`. The definition for `is-lon?` defines the projection function that tests the argument given to `addlist`. If it is a list of numbers then no error is raised. Otherwise, a blame error is raised that requires as input the blame object, the value protected, and a customized message (in our example, a string containing the name of the function followed by a colon). The formatting function given to `current-blame-format` ignores the blame object and constructs an error message using the other two arguments. Evaluating `(addlist '(1 2 c))` yields:

```
addlist:  expected a list of numbers but got (1 2 c)
```

As the reader may appreciate, the error message is concise and informative. The user can clearly see that a list of numbers is not given to `addlist`. Clearly, this is better than throwing an error when applying `+` in the contractless version.

The assignment of blame and customizable error messages makes contracts ideal to implement a tailored-made error-messaging system. In this article, we put forth a contract-based error-messaging system designed and implemented for **FSM**—a domain-specific language for the automata theory classroom embedded in **Racket** [11]. Unlike **FSM**'s previous error-messaging system, which is built on the use of exceptions and bulk messages for all found errors, the new error-messaging system presents the programmer with short well-focused messages for one argument at a time. In addition, the new error-messaging system is tightly-coupled with a design recipe [1,9,10] for state machines. Every error message indicates which step of the design recipe has not been successfully completed. Thus, one of the contributions of this article is to illustrate that contracts are well-suited to support a program-by-design methodology. The error-messaging system, in part, plays the role of a tutor that provides immediate feedback. As is

well-known, providing immediate feedback and comprehensible error messages is essential in a learning context [14]. In addition, the error messages are crafted with the important characteristic of not being prescriptive [5,6]. That is, they do not suggest to the programmer the solution to the error detected. Another contribution of this article is to combine contracts and testing—an essential step in any design recipe. The programmer, for example, may provide a list of words that ought to be accepted and a list of words that ought to be rejected by the constructed machine. If there is a failure for any word in either list then an error is thrown and the machine constructor fails.

The article is organized as follows. Section 2 provides a brief overview of programming and designing in **FSM**. Section 3 discusses **FSM** contracts for machine constructors. Section 4 reviews how testing is incorporated with contracts. Section 5 compares and contrasts related work with the work presented in this article. Finally, concluding remarks and directions for future work are presented in Section 6.

2 Design Using **FSM**

In **FSM**, programmers can construct state machines, grammars, and regular expressions. The focus in this article is on state machine construction: deterministic finite-state automata (**dfa**), nondeterministic finite-state automata (**ndfa**), push-down automata (**pda**), Turing machines (**tm**)s, and multitape Turing machines (**mttm**)s. This section first presents a brief overview of state machine construction in **FSM**. Then it reviews the design recipe for state machines and ends with a small illustrative example.

2.1 State Machine Construction in **FSM**

The values needed to construct state machines in **FSM** are described as follows:

K: A list of states, where each state is a symbol representing an uppercase letter in the Roman alphabet or an uppercase letter in the Roman alphabet followed by a - and a natural number (no blank characters allowed).

Σ: A list of symbols representing an input alphabet, where each symbol represents a lowercase letter in the Roman alphabet.

Γ: A list of symbols representing a stack alphabet, where each symbol represents a letter in the Roman alphabet.

S: The starting state such that $S \in K$.

F: A list of final states such that $F \subseteq K$

R: A list of transition rules.

Y: An optional argument that denotes the accept state for a Turing machine or multitape Turing machine that decides or semidecides a language such that $Y \in K$.

add-dead: A Boolean to indicate if a dead state and missing transitions be added to complete the transition function for a deterministic finite state automaton.

num-tapes: A natural number for the number of tapes in a multitape Turing machine.

The state machine constructors are:

dfa: (make-dfa K Σ S F R [add-dead]), where R is a **dfa** transition function.
ndfa: (make-ndfa K Σ S F R), where R is an **ndfa** transition relation.
pda: (make-ndpda K Σ Γ S F R), where R is an **pda** transition relation.
tm: (make-tm K Σ R S F [Y]), where R is an **tm** transition relation.
mttm: (make-mttm K Σ S F R num-tapes [Y]), where R is an **mttm** transition relation

For **tm** and **mttm**, the Y argument is optional because such machines may not decide or semidecide a language. The natural number given to a **mttm** is for the number of tapes.

The transition rule type depends on the machine constructed. The signatures for transition rules are:

dfa: (K Σ K), where in the first state the input alphabet element is read and the machine transitions to the second state.
ndfa: (K $\Sigma \cup \{\text{EMP}\}$ K), where in the first state the input alphabet element or nothing is read, and the machine transitions to the second state.
pda: ((K $\Sigma \cup \{\text{EMP}\}$ (listof Γ)) (K (listof Γ))), where in the first state the input alphabet element or nothing is consumed and the first list of Γ elements are popped before moving to the second state and pushing the second list of Γ elements.
tm: ((K $\Sigma \cup \{\text{LM BLANK}\}$) (K $\Sigma \cup \{\text{LEFT RIGHT BLANK}\}$)), where in the first state the tape element is read and the machine transitions to the second state writing to the tape or moving the head.
mttm: ((list K (listof $\Sigma \cup \{\text{LM BLANK}\}$)) (list K (listof $\Sigma \cup \{\text{LEFT RIGHT BLANK}\}$))), where in the first state the input elements on each tape are read and the machine transitions to the second state writing to or moving the head on each tape.

Here, **BLANK** and **LM** are FSM constants, respectively, for a blank and for the left-end marker.

A subset of observers are:

```
(sm-states M) (sm-sigma M) (sm-start M) (sm-finals M) (sm-rules M)
(sm-type M)
(sm-apply M w)
(sm-showtransitons M w)
(sm-test M [N])
```

The first 5 observers extract a component from the given state machine. The given state machine's type is returned by (sm-type M). To apply a given machine to the given word (sm-apply M w) is used. It returns 'accept or 'reject. A trace of the configurations traversed by applying a given machine to a given word is obtained using (sm-showtransitons M w). Finally, sm-test applies the given machine to n randomly generated words and returns a list of the words and the obtained results.

1. Name the machine and specify alphabets
2. Write unit tests
3. Identify conditions that must be tracked as input is consumed, associate a state with each condition, and determine the start and final states.
4. Formulate the transition relation
5. Implement the machine
6. Test the machine using unit tests and random testing
7. Design, implement, and test an invariant predicate for each state
8. Prove $L = L(M)$

Fig. 1: The design recipe for state machines.

2.2 Design Recipe for State Machines

The design recipe for state machines presented to students is displayed in Figure 1. Step 1 asks to pick a descriptive name for the machine and to define the needed alphabets (i.e., input and, if applicable, stack alphabets).

Step 2 asks for the development of unit tests. These must include tests for words that are accepted and for words that are rejected.

Step 3 asks to associate a state with each condition of the consumed input (and, if applicable, the stack) that must be tracked as the machine consumes a word. This includes choosing valid state names. In addition, this state asks to identify the start and final states. These states must be in the set of states identified to track conditions.

Step 4 asks for the development of the transition relation. Only alphabet elements identified in Step 1, states identified in Step 3, and, if appropriate, the FSM constants for a blank, the empty symbol, and the left-end marker may be used to formulate the transition relation. Students develop each transition assuming that the condition describing the source state holds and aim that the action(s) taken by the machine make the conditions of the destination state hold.

Steps 5 and 6 ask for the machine's implementation and the running of unit and random tests.

Step 7 asks for the development of state invariant predicates to validate that the condition each state represents holds during machine execution. These are used to validate the machine during execution using the FSM visualization tool [12].

Finally, Step 8 asks for the development of a proof demonstrating that the machine's language, $L(M)$, is the same as the language, L , the machine is designed to decide. This is done by first proving by induction that the state invariants hold when an arbitrary word is processed. This proof is then used to prove that $L = L(M)$.

2.3 Illustrative Example

To illustrate the steps of the design recipe in action, consider designing and implementing a pda for $L = a^n b^{2^n}$. The design idea is to push two a s onto the

```

#lang fsm

;; L = anb2n
(define P (make-ndpda '(S F)
  '(a b)
  '(a)
  'S
  '(F)
  `(((S a ,EMP) (S (a a)))
    ((S ,EMP ,EMP) (F ,EMP))
    ((F b (a)) (F ,EMP)))))

(check-equal? (sm-apply P '(b a)) 'reject)
(check-equal? (sm-apply P '(a a a)) 'reject)
(check-equal? (sm-apply P '(a b b a a b b b b)) 'reject)
(check-equal? (sm-apply P '()) 'accept)
(check-equal? (sm-apply P '(a a b b b b)) 'accept)

```

Fig. 2: A pda for $L = a^n b^{2n}$.

stack for each a at the beginning of the input word. Once a s are pushed, for every b read an a is popped.

The results for Step 1 are:

```

Name: P
Σ = '(a b)
Γ = '(a)

```

Observe that each alphabet only contains valid FSM alphabet elements.

The following unit tests are written to satisfy Step 2:

```

(check-equal? (sm-apply P '(b a)) 'reject)
(check-equal? (sm-apply P '(a a a)) 'reject)
(check-equal? (sm-apply P '(a b b a a b b b b)) 'reject)
(check-equal? (sm-apply P '()) 'accept)
(check-equal? (sm-apply P '(a a b b b b)) 'accept)

```

Observe that the tests include both accepted and rejected words.

The documentation required for the states in Step 3 is summarized as follows:

```

S: consumed input = stack = a*, starting state
F: consumed input = anb2n-|stack| ∧ stack = a*, final state

```

The required transition relation for Step 4 is:

```

`(((S a ,EMP) (S (a a)))
  ((S ,EMP ,EMP) (F ,EMP))
  ((F b (a)) (F ,EMP)))

```

In state **S** twice as many **a**s as found at the beginning of the word are pushed onto the stack. The machine nondeterministically decides to move to state **F** and matches **b**s with **a**s on the stack. If all **b**'s are matched and the stack is empty then the machine accepts. Observe that in such a case the invariant condition for **F** informs us that the consumed input (i.e., the entire input word) is of the form $a^n b^{2n}$.

The implementation of the machine to satisfy Step 5 is displayed in Figure 2. For Step 6, running the tests reveals that they all pass. In the interest of brevity, we omit Steps 7 and 8 given that they are not directly involved in code development.

3 FSM Contracts

This section outlines machine-constructor contracts. First, the contracts for non-dependent types, like states and alphabets, are outlined. Second, the contracts for dependent types, like final states and transition rules, are outlined. In general, the structure of the machine-constructor contracts is as follows:

```
(->i ([states <contract>]
      [sigma <contract>]
      [gamma <contract>] ;; only for ndpda
      [start (<dependencies>) <contract>]
      [finals (<dependencies>) <contract>]
      [rules (<dependencies>) <contract>]
      [num-tapes <contract>]) ;; only for mttm
      ([add-dead <contract>] ;; only for dfa
       [accept (<dependencies>) <contract>] ;; only for tm/mttm
       #:accepts [accepts (<dependencies>) <contract>]
       #:rejects [rejects (<dependencies>) <contract>])
      [result <contract>])
```

Here, `->i` is a contract combinator that allows expressing the dependencies between arguments and results [3, Section 8.2]. Each argument and each result may be named and, subsequently, used in subcontracts. It handles required, optional, and keyword parameters. For each argument, there is a stanza containing the argument's name, an optional lists of dependencies, and a contract that must be satisfied.

To briefly outline the syntax, consider the header for the `dfa` constructor:

```
(define/contract (make-dfa states sigma start finals rules
                          [add-dead #t]
                          #:accepts [accepts '()])
  #:rejects [rejects '()])
```

There are 5 required parameters, 1 optional parameter, and two keyword parameters. For a non-dependent required parameter, like `states`, only the name and the contract for the condition are needed. For instance, the stanza for `states` may be outlined as follows:


```
[states (and/c ...)]
```

Here, `and/c` is a contract combinator that takes as input an arbitrary number of contracts and that accepts any value that satisfies all the given contracts [3, Section 8.2]. The provided contracts are applied from left to right (or top to bottom) and, if any, the first failing contract attributes the blame for failure. For a dependent argument, there is the name, the dependencies, and the contract for the condition. For example, the stanza for `finals` may be outlined as follows:

```
[finals (states) (and/c ...)]
```

This stanza states that the value of `finals` depends on `states`. The contracts provided to `and/c` may refer to states. Finally, contracts for optional and keyword parameters are listed separately from the required parameters. For keyword parameters, the keyword must appear before its contract stanza. For instance, the stanza for a list of words that ought to be accepted may be outlined as follows:

```
#:accepts [accepts (states sigma start finals rules add-dead)
            (and/c ...)]
```

The list of words that ought to be accepted depends on all required and optional arguments given that the machine must be constructed to be applied to each word. We note that if the contract for `accepts` is evaluated then it is safe to build the machine given that the contracts for the required and optional arguments are satisfied.

3.1 States and Alphabet Contracts

These contracts test that the argument is a list, that each element is valid (as defined in Section 2.1), and that there are no duplicates. Therefore, 3 (sub)contracts are employed. Their stanzas are implemented as follows for all machine types:

```
[states (and/c (is-a-list/c "machine states" "three")
               (valid-listof/c valid-state?
                               "machine state"
                               "list of machine states"
                               #:rule "three")
               (no-duplicates/c "states"))]

[sigma (and/c (is-a-list/c "machine alphabet" "one")
              (valid-listof/c valid-alpha?
                              "alphabet letter"
                              "input alphabet"
                              #:rule "one")
              (no-duplicates/c "sigma"))]
```

Here, `is-a-list/c` is a flat contract (i.e., a predicate) that checks the provided argument is a list. It takes two string arguments, identifying the machine component and the design step, that are used to build the error message when the contract fails. The flat contract `valid-listof/c` takes as input a predicate to apply to each list element and three strings, to build the error message when the predicate is not satisfied, identifying the element that fails, the argument that fails, and the design recipe rule that is not successfully completed. In addition, it accumulates all the elements that do not satisfy the predicate for the error message. Finally, the last flat contract checks for duplicates and takes as input a string identifying, if necessary, the argument whose contract fails.

To illustrate the states and sigma contracts, consider the following use of the constructor for `dfa`:

```
(make-dfa '(A B B C A)
          '(a b c D 1)
          'A
          '(B C)
          `((A b C)
            (A c C)
            (B c B)
            (B a B))
          #t)
```

When the expression is evaluated, the reported error is:

```
Step three of the design recipe has not been successfully
completed. There following values, (A B), are duplicated
in the given states: (A B B C A)
```

Observe that the accumulated repeated values and the given state list are clearly identified along with the design recipe step not successfully completed. Upon correcting the state list and calling the `dfa` constructor again, the following error is reported:

```
Step one of the design recipe was not successfully
completed. The following: (D 1) are not valid
lowercase alphabet letters in the given input
alphabet: (a b c D 1)
```

Observe that all the elements that do not represent lowercase Roman letters in the given argument are clearly identified. Upon correcting this error, all the (sub)contracts are satisfied and the machine is constructed.

3.2 Start and Final States

For all machine constructors, the contract for the start state checks that the argument is a valid start state (i.e., a symbol representing an uppercase Roman alphabet letter that is in the given list of states). It's contract is implemented as follows:

```
[start (states) (and/c (valid-start/c states)
                        (start-in-states/c states))]
```

Technically, it suffices to check if the argument for `start` is in the given state list. Concerns are separated into two contracts to provide better error messages: the argument is a valid state and the argument is in the state list. For instance, consider the following use of the constructor for `pdas`:

```
(make-ndpda ' (S M F)
             ' (a b)
             ' (a)
             ' (S-1)
             ' (F)
             `(((S ,EMP ,EMP) (M ,EMP))
               ((S a ,EMP) (S (a)))
               ((M b (a)) (M ,EMP))
               ((M ,EMP ,EMP) (F ,EMP)))))
```

The following error message is generated:

```
Step three of the design recipe was not successfully
completed. The given starting state: (S-1) is not a
valid state
```

In addition to clearly identifying the design recipe step that is not successfully completed, the message identifies the argument as an invalid state. A correction attempt changes the type of the argument to, `'S-1`, a state symbol. Upon running the constructor again, the following error message is generated:

```
Step three of the design recipe has not been successfully
completed. The following starting state, S-1, is not in the
given list of states: (S M F)
```

Observe that the message is now different. It emphasizes that the error detected is related to the dependency of the starting state argument. Upon providing an element in the state list as the argument, the contract is satisfied and the machine is constructed.

Testing the argument for the final states is done the same way for all machines. The contract determines that the argument is a list, that all the list elements are valid states, that all the list elements are in the state list, and that there are no repetitions. The contract is implemented as follows:

```
[finals (states)
  (and/c (is-a-list/c "machine final states" "three")
    (valid-listof/c valid-state?
      "machine state"
      "list of machine finals"
      #:rule "three")
    (valid-finals/c states)
    (no-duplicates/c "final states"))]
```

It reuses several of the flat contracts discussed above. To illustrate its use consider the following attempt to construct a `tm`:

```
(make-tm '(S Y N)
  `(a b)
  `(((S a) (S ,RIGHT))
    ((S b) (N b))
    ((S ,BLANK) (Y ,BLANK))))
'S
'(Y M)
'Y)
```

The following error is reported:

```
Step three of the design recipe has not been successfully
completed. The following final states, (M), are not in your
list of states: (S Y N)
```

Once again, the design recipe step violated is clearly identified and, in this case, the error focuses on the dependency for the final states. Changing `M` to `S` or `N` satisfies the contract and the machine is constructed.

3.3 Transition Rules

The contracts for transition rules are customized for each machine type given that their transition rules have different dependencies and different structures. They all, however, share common design features. Each of these contracts checks that the argument is a list, that each list element has the correct transition structure, that each transition respects the dependency of the states and the input alphabet, and that the list contains no duplicates. Figure 3 displays the contract implementations for transition rules. Observe that there are differences in the contracts for `dfa`, `pda`, and `mttm` transition rules. For `dfa` transitions, there is a dependency on `add-dead` that is used to test if the given argument denotes a function. For `pda` transitions, there is a dependency on the stack alphabet that is used to test that all nonempty pop and push elements are members of the `gamma` argument. Finally, for `mttm` transitions, there is a dependency on the number of tapes that is used verify that each transition contains the correct number of actions (i.e., one for each tape). The flat (sub)contracts combined are straightforward to implement and in the interest of brevity are not presented.

To illustrate a transition relation contract in action, consider building a 3-tape `mttm` to add two numbers in unary notation as displayed in Figure 4. The first reported error informs the programmer that the first transition rule is not properly structured:

```
Step four of the design recipe was not successfully completed.
The following rules have structural errors:
Rule ((S ( _ _ _)) (A (R R))):
  The second element in the second part of the rule, (R R),
  is not a list of 3 symbols.
```

```

;; dfa
[rules (states sigma add-dead)
  (and/c (is-a-list/c "machine rules" "four")
    correct-dfa-rule-structures/c
    (correct-dfa-rules/c states sigma)
    (functional/c states sigma add-dead)
    (no-duplicates-dfa/c "rules"))]

;; ndfa
[rules (states sigma)
  (and/c (is-a-list/c "machine rules" "four")
    correct-dfa-rule-structures/c
    (correct-dfa-rules/c states (cons EMP sigma))
    (no-duplicates/c "rules"))]

;; pda
[rules (states sigma gamma)
  (and/c (is-a-list/c "machine rules" "four")
    correct-ndpda-rule-structures/c
    (correct-ndpda-rules/c states sigma gamma)
    (no-duplicates/c "rules"))]

;; tm
[rules (states sigma)
  (and/c (is-a-list/c "machine rules" "four")
    correct-tm-rule-structures/c
    (correct-tm-rules/c states sigma)
    (no-duplicates/c "rules"))]

;; mttm
[rules (states sigma num-tapes)
  (and/c (is-a-list/c "machine rules" "four")
    (correct-mttm-rule-structures/c num-tapes)
    (correct-mttm-rules/c states sigma)
    (no-duplicates/c "rules"))]

```

Fig. 3: Contracts for transition rules.

In this case, there is a missing action, R, for one of the tapes. Upon adding it, the following error is reported is:

```

The following rules have errors, which make them invalid:
Rule ((T (_ _ _)) (Q (L _ _))):
  The to state, Q, is not in the given list of states.
Rule ((Q (_ _ _)) (U (L L L))):
  The from state, Q, is not in the given list of states.
Rule ((U (_ i _)) (V (_ I _))):
  The action I on tape 1 must be in the given input alphabet,

```

```

(make-mttm '(S A T U V H)
  ~(i)
  'S
  'H)
  (((S (,BLANK ,BLANK ,BLANK)) (A (R R)))
   ((A (i ,BLANK ,BLANK)) (A (,BLANK i ,BLANK)))
   ((A (,BLANK i ,BLANK)) (A (R R ,BLANK)))
   ((A (,BLANK ,BLANK ,BLANK)) (T (R ,BLANK ,BLANK)))
   ((T (i ,BLANK ,BLANK)) (T (,BLANK ,BLANK i)))
   ((T (,BLANK ,BLANK i)) (T (R ,BLANK R)))
   ((T (,BLANK ,BLANK ,BLANK)) (Q (L ,BLANK ,BLANK)))
   ((Q (,BLANK ,BLANK ,BLANK)) (U (L L L)))
   ((U (,BLANK i i)) (U (i i ,BLANK)))
   ((U (i i ,BLANK)) (U (L i L)))
   ((U (,BLANK i ,BLANK)) (V (,BLANK I ,BLANK)))
   ((V (,BLANK i ,BLANK)) (V (i ,BLANK ,BLANK)))
   ((V (i ,BLANK ,BLANK)) (V (L L ,BLANK)))
   ((V (,BLANK ,BLANK ,BLANK)) (H (,BLANK ,BLANK ,BLANK))))
  3)

```

Fig. 4: A proposed `mttm` to add two natural numbers in unary notation.

LEFT, RIGHT, or BLANK.

These are two dependencies errors on the states and the input alphabet. Upon adding Q to the state list and changing I to i, the `mttm` is successfully constructed.

3.4 Other Flat Contracts

The remaining two needed flat contracts are fairly straightforward predicates. One is for `tms` that are language recognizers. For these machines, an accept state must be provided. This state must be a valid state, not be FSM's default DEAD state, and be in the list of final states.

For `mttms`, a natural number greater than 0 must be provided for the number of tapes.

4 Contracts for Testing

To assist programmers to properly validate machines, we use contracts to provide a list of words that ought to be rejected and a list of words that ought to be accepted. These are implemented as keyword parameters whose default value is the empty list. The contracts for these parameters are the last listed and, therefore, only report an error, if any, after the contracts for all other machine components are satisfied. This approach results in two advantages. The first, this mechanism may be used to collect words for which the wrong result is obtained

by previous testing. The second, it can cut down the amount of coding required by the programmer. In lieu of writing unit tests, a programmer simply lists the words that ought to be tested in the proper list.

For all machine types, the contract follows the same general design based on three (sub)contracts. The first tests that the argument is a list of words and takes as input a string identifying if the test is for accepts or rejects. The second tests that the words consist of elements in the input alphabet. The third, builds the machine and accumulates, if any, words that do not produce the expected result to build the error message. It takes as input a symbol identifying if words ought to be accepted or rejected. We note that building the machine is safe given that the contracts on the machine components are satisfied. For example, the `#accepts` contract for `ndfa` is implemented as follows:

```
#:accepts
[accepts
 (states sigma start finals rules)
 (and/c (listof-words/c "accepts")
 (words-in-sigma/c sigma)
 (ndfa-input/c states sigma start finals rules 'accept))]
```

The contract for `#rejects` is build in a similar fashion substituting "accepts" with "rejects" and 'accept with 'reject.

To illustrate the contract behavior, consider the following attempt to build a dfa:

```
(make-dfa '(A B)
           '(a b)
           'A
           '(A)
           '((A a A)
             (B b B)
             (A b B)
             (B a B))
           #t
           #:accepts '((a a a a a) (b))
           #:rejects '((a a a) (a a a a b a)))
```

Upon evaluating this expression, the following error is reported:

```
Step six of the design recipe has not been successfully
completed. The constructed machine does not accept the
following words: ((b))
```

Observe that the error clearly indicates that testing (Step six of the design recipe) has not been properly completed. In this case, a word that ought to be rejected appears in the `accepts` argument. Upon moving '(b) to the `rejects` argument and reevaluating, the following error is reported:

Step six of the design recipe has not been successfully completed. The constructed machine does not reject the following words: ((a a a))

Once again, we see that the design recipe's testing step has not been successfully completed. In this case, a word that ought to be accepted appears in the **reject** argument. Upon moving this word to the **accepts** argument, the machine is successfully constructed.

There are small variations in the contracts for some of the other machine types. For instance, the **#accepts** and **#rejects** contracts for **tms** test that the initial head position is a valid index into the given tape. In addition, the given tape is allowed to contain blanks. The same is true for the **mttm** contract.

5 Related Work

5.1 Design Recipe

A design recipe is a series of steps that defines a systematic process to implement functions. Each step has a specific outcome and brings a programmer closer to the solution to a problem expressed as a program. They are extensively used in curricula for first-year students [1,9,10]. Such curricula start by emphasizing type-driven design. That is, the structure of the data guides the development of solutions. Such an approach leads naturally to structural recursion and functional abstraction. Surprisingly, a common misconception is that design recipes are limited problem solving based on structure. Nothing is farther from the truth. Design recipes are also used to design solutions based on insights to a problem (i.e., generative recursion), on avoiding the loss of knowledge (i.e., accumulative recursion), and on mutation. As a consequence, this means that design recipes can (and ought) to be vertically integrated into the Computer Science curriculum. The work done with **FSM** is an attempt to introduce systematic design into the Formal Languages and Automata Theory classroom [11,12].

In our experience, however, a design recipe is not enough for the bulk of students in upper-level courses. The top students usually are disciplined and make sure to successfully complete all the steps of the design recipe. Other students are less disciplined. They may, for example, find a bug through unit testing and delete (or comment out) failing tests. Unfortunately, programmers get distracted and when they return to their program haven't forgotten about the bug. When they run their code all the unit tests pass and consider the assignment ready for grading. This leads to lower marks and clearly indicates that the language ought to provide more help. To this end, the contract-based error-messaging system described in this article allows programmers to provide a constructor with a list of words that have failed to be accepted and a list of words that have failed to be rejected. This provides the programmer with a guarantee that machine construction fails until the machine is correctly implemented to accept or reject the listed words. Thus, students that delete or comment out unit tests that fail will be reminded that they have not yet successfully designed their machine.

5.2 Contracts

Given that design recipes encompass type-driven design, contracts are a natural partner. For instance, in FSM machine constructors are defined in modules that define the language. The former FSM error-messaging system threw an error on constructor misuse (e.g., providing a wrong type of argument). Unlike traditional error-messaging systems that provide a single error, multiple errors were provided at once partitioned in two by error type. The first partition reported errors on arguments that do not depend on other arguments (e.g., the names of the states or the elements of the input or stack alphabets). The second reported errors on the dependent types (e.g., the transition rules). The goal was to provide an informative global view of constructor misuse. For instance, consider the following misuse of the `dfa` constructor:

```
(make-dfa '(A B C)
          '(a b c)
          'a
          '(B C D)
          '((A b D)
            (A a c)
            (B b B)
            (B a c))))
```

The error reported by the former error-messaging system is:

```
-STARTING states CONTENT ERROR:
  Starting states a is not in the list of states (A B C)
-FINAL STATES CONTENT ERRORS:
  The final states (D) are not contained in the list of
  states (A B C)
-RULE CONTENT ERRORS:
  The following symbols are not defined in your list of
  states or alphabet: (D).
- THE RHS OF THE FOLLOWING RULES ARE NOT VALID:
  (A b D)
  (A a c)
  (B a c)
Check above message for errors
```

Observe that errors for multiple arguments are reported at once. In practice, such an approach revealed significant drawbacks. This approach works well for top students that take the time to read and understand all the error messages. Other students, unfortunately, are frustrated by the length of the error message and, typically, interpret that all the error messages are related to each other (when they are not). Clearly, an overhaul of FSM's error-messaging system was needed for it to be effectively useful for more students. The work described in this article uses contracts in such an overhaul. To this end, the work presented exchanges "long" global view error messages with shorter error messages that

are reported one at a time for each argument provided to a constructor. First, errors, if any, are reported (one at a time) for non-dependent types (e.g., states and alphabets). Second, errors are reported (one at a time) for dependent types (e.g., start state, final states, and transition rules). Furthermore, for errors in the transition rules, a single rule is processed at a time. In this manner, the average student is focused to fix errors involving a single argument one at a time starting with the non-dependent types. In addition, the error message generated serves as a tutor indicating the step of the design recipe that must be revisited. For instance, if a programmer uses an invalid state name the blame message is tailored to communicate which state name is invalid and to specify that the design recipe step for naming states has been violated. We note that the error messages are useful for programmers that use **FSM** without knowledge of the associated design recipes.

Racket provides support for 3 varieties of contracts: flat, chaperone, and impersonator [3, Chapter 8]. Flat contracts are fully checked upon receiving a value and are, essentially, predicates. Chaperone contracts wrap around a value and do not determine a violation until the value is used. For instance, chaperone functions are used for function arguments and are not evaluated until the chaperoned function is used. Impersonator contracts are wrapped around a value and redirect some of its operations. The work presented in this article only uses flat contracts. Arguments and optional arguments to machine constructors as well as the machine type returned are immediately checked upon receiving input. Currently, **FSM** machine constructors do not consume argument functions and, therefore, we do not face the potential pitfalls addressed by contracts for higher-order functions [2].

Finally, **Racket** provides support for keyword arguments. These arguments are not passed by position in a function's parameter list. Instead, they are passed using a keyword. A keyword argument must always have a value, but if a default value is provided then the programmer does not have to provide an argument for it. When a default value is provided, the programmer may think of a keyword argument as optional. **FSM** constructor-contracts exploit keyword arguments to integrate the testing step of the design recipe. To build a machine, programmers may or may not provide a list of words that are and a list of words that are not in the machine's language. An **FSM** machine-constructor contract detects if a word that ought to be accepted is rejected and if a word that ought to be rejected is accepted. When this occurs, an informative error message is generated and the constructor fails. To the best of our knowledge, this is the first time that contracts and testing have been combined as a single system.

6 Concluding Remarks

This article presents a novel contract-based error-messaging system for, **FSM**, a domain specific language embedded in **Racket** for the automata theory classroom. The described system has well-known desirable features such as concise, understandable, and nonprescriptive error messages. In addition, the system de-

scribed implements two novel features. The first is that it provides support for a program by design methodology that uses a design recipe. Every error message generated indicates the design recipe step that has not been successfully completed. The second is that contracts are used to integrate testing into the error-messaging system. Programmers may provide lists of words that ought to be accepted and ought to be rejected. Machine construction fails if for any provided word the wrong result is obtained.

Future work is two-fold. On the one hand, we wish to run several iterations of a class using the described error-messaging system and collect empirical data from the students. We will measure student perceptions as well as the usefulness of the error messages. On the other hand, we will extend the described system to grammar constructors (i.e., regular, context-free, and context-sensitive) and to regular expression constructors.

References

1. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: How to Design Programs: An Introduction to Programming and Computing. MIT Press, Cambridge, MA, USA, Second edn. (2018)
2. Findler, R.B., Felleisen, M.: Contracts for Higher-Order Functions. In: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming. p. 48–59. ICFP '02, Association for Computing Machinery, New York, NY, USA (2002). <https://doi.org/10.1145/581478.581484>, <https://doi.org/10.1145/581478.581484>
3. Flatt, M., PLT: The Racket Reference, <https://docs.racket-lang.org/reference/index.html>, last accessed: November 2023
4. Lazarek, L., Greenman, B., Felleisen, M., Dimoulas, C.: How to Evaluate Blame for Gradual Types, Part 2. Proc. ACM Program. Lang. **7**(ICFP) (aug 2023). <https://doi.org/10.1145/3607836>
5. Marceau, G., Fisler, K., Krishnamurthi, S.: Measuring the effectiveness of error messages designed for novice programmers. In: Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education. pp. 499–504. SIGCSE '11, ACM, New York, NY, USA (2011). <https://doi.org/10.1145/1953163.1953308>
6. Marceau, G., Fisler, K., Krishnamurthi, S.: Mind your language: On novices' interactions with error messages. In: Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. pp. 3–18. Onward! 2011, ACM, New York, NY, USA (2011). <https://doi.org/10.1145/2048237.2048241>
7. Meyer, B.: Advances in Object-Oriented Software Engineering, chap. Design by Contract, pp. 1–50. Object-Oriented series, Prentice Hall (1992)
8. Meyer, B.: Applying Design by Contract. Computer **25**(10), 40–51 (oct 1992). <https://doi.org/10.1109/2.161279>
9. Morazán, M.T.: Animated Problem Solving - An Introduction to Program Design Using Video Game Development. Texts in Computer Science, Springer (2022). <https://doi.org/10.1007/978-3-030-85091-3>
10. Morazán, M.T.: Animated Program Design - Intermediate Program Design Using Video Game Development. Texts in Computer Science, Springer (2022). <https://doi.org/10.1007/978-3-031-04317-8>

11. Morazán, M.T., Antunez, R.: Functional automata-formal languages for computer science students. In: Caldwell, J.L., Hölzenspies, P.K.F., Achten, P. (eds.) Proceedings 3rd International Workshop on Trends in Functional Programming in Education, TFPIE 2014, Soesterberg, The Netherlands, 25th May 2014. EPTCS, vol. 170, pp. 19–32 (2014). <https://doi.org/10.4204/EPTCS.170.2>, <https://doi.org/10.4204/EPTCS.170.2>
12. Morazán, M.T., Schappel, J.M., Mahashabde, S.: Visual Designing and Debugging of Deterministic Finite-State Machines in FSM. *Electronic Proceedings in Theoretical Computer Science* **321**, 55–77 (aug 2020). <https://doi.org/10.4204/eptcs.321.4>
13. Plösch, R.: Design by Contract for Python. In: 4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC '97 / ICSC '97), 2-5 December 1997, Clear Water Bay, Hong Kong. pp. 213–219. IEEE Computer Society (1997). <https://doi.org/10.1109/APSEC.1997.640178>, <https://doi.org/10.1109/APSEC.1997.640178>
14. Race, P.: Using Feedback to Help Students Learn. <https://www.heacademy.ac.uk/knowledge-hub/using-feedback-help-students-learn> (January 2005)