TRACECONTRACT: A Scala DSL for Trace Analysis[⋆]

Howard Barringer[1] and Klaus Havelund[2]

[1] School of Computer Science, University of Manchester, UK
Howard.Barringer@manchester.ac.uk

[2] Jet Propulsion Laboratory, California Institute of Technology, USA
Klaus.Havelund@jpl.nasa.gov

Abstract. In this paper we describe TRACECONTRACT, an API for trace analysis, implemented in the SCALA programming language. We argue that for certain forms of trace analysis the best weapon is a high level programming language augmented with constructs for temporal reasoning. A trace is a sequence of events, which may for example be generated by a running program, instrumented appropriately to generate events. The API supports writing properties in a notation that combines an advanced form of data parameterized state machines with temporal logic. The implementation utilizes SCALA's support for defining internal Domain Specific Languages (DSLs). Furthermore SCALA's combination of object oriented and functional programming features, including partial functions and pattern matching, makes it an ideal host language for such an API.

1 Introduction

The trace analysis problem consists of determining whether a trace, a sequence of events, satisfies a formalized property. One challenge is to find convenient and expressive languages for expressing such trace properties. We present in this paper an API, named TRACECONTRACT, in the SCALA programming language [2] for performing trace analysis (runtime verification). It supports writing temporal properties about traces and can be used for analyzing log files produced as a result of program executions or for monitoring systems executing online. The contribution of the paper is a convenient and very expressive specification notation, which can be perceived as a hybrid between state machines and temporal logic, but formulated as an API in a high level programming language. This allows a mixture of temporal specification and high level programming, a combination we find very attractive for practical purposes. The implementation of the API benefits from SCALA's support for defining domain specific languages and from its functional programming features. This includes specifically the use of partial functions and pattern matching over parameterized events to model state transitions, which is very similar to the way to the `receive` function in SCALA's `Actor` class is implemented to model an actor's reception of messages from other concurrently running actors. The API and SCALA have been chosen for analysis of command

[⋆] Part of the research described in this publication was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

sequences for NASA's LADEE (Lunar Atmosphere and Dust Environment Explorer) mission [1].

A large number of formalisms have been proposed in recent years for supporting trace analysis, see for example [13, 9, 12, 10, 3, 15, 8]. Examples are temporal logics, including past time as well as future time, regular expressions, state machines, context free grammars, real-time logics, and statistics gathering logics. Most have been implemented as what are often referred to as *external DSLs* (Domain Specific Languages), external to the programming language they are implemented in, and parsed with a specialized parser. Our own work includes several such systems, most of which put emphasis on expressiveness, in order to be able to capture the many different logics provided in other systems. Amongst these systems are EAGLE [4] — based on recursive definitions of temporal predicates; RULER [7] — a rule-based framework providing the same level of formal expressivity as EAGLE but with a simpler and more efficient step-wise monitoring algorithm; and LOGSCOPE [5] — a state machine-like subset of RULER with the addition of a temporal logic. From these experiences, we observe two key points: (i) once a DSL is defined, it is laborious to change/extend it later; and (ii) users often ask for additional features, some of which are best handled by a general purpose programming language. We propose here instead to write trace monitors in a high level programming language, SCALA, augmented with support for temporal specification. Our solution is what is referred to as an *internal DSL*, internal to (embedded in) the programming language it is developed in. Stolz and Huch describe in [16] an embedding of LTL in HASKELL. Our framework differs in two major ways. First, we handle data parameterization by re-using SCALA's built-in notion of partial functions and pattern matching. Second, we introduce a new formalism which is a hybrid between state machines and temporal logic.

TRACECONTRACT is really just an API, formulated using the host language's primitives. It does, however, have the flavor of a DSL due to SCALA's special support for defining internal DSLs, and due to the fact that SCALA supports functional as well as object oriented programming. We shall use the terms API and DSL interchangeably. The DSL is a *shallow* embedding, meaning that we are making the host language's constructs part of the DSL. This is in contrast to a *deep embedding*, as in [16], where a separate internal representation is made of the DSL (an abstract syntax), which is then interpreted or compiled as in the case of an external DSL. See [11] for a recent discussion of shallow vs. deep embeddings. Generally, the arguments *for* a shallow internal DSL are: limited implementation effort (leading to adaptability), feature richness through the host language, and tool inheritance. The arguments *against* are: lack of analyzability — which can have consequences for performance and reporting to users, and full exposure of the implementation language (the user has to be a programmer, in this case a SCALA programmer). The groundwork for a theoretical study of the characteristics of the approach, such as soundness, completeness, and expressiveness, has been done during our previous work on EAGLE and RULER. However, a full theoretical presentation of TRACECONTRACT, including a formal semantics, is planned.

The rest of the paper is organized as follows. Section 2 presents the API and examples of its use. Section 3 outlines how the API is implemented. Section 4 concludes the paper.

2 The TraceContract DSL

A trace contract conceptually represents a predicate on execution traces, where an execution trace is a finite sequence of events. TRACECONTRACT is parameterized with the event type, which may be any type. To illustrate, we shall consider a simplified planetary rover scenario (to be our on-going example), similar to the example used in [5][3]. A rover is controlled from ground via commands emitted to it. Commands can either fail or succeed. Events are commonly modeled as objects (instances) of certain classes. The following SCALA classes define the type `Event` of events, and three specific kinds of events we are interested in monitoring:

Listing 1.1. Type Event

```
abstract class Event
case class COMMAND(name: String, nr: Int) extends Event
case class SUCCESS(name: String, nr: Int) extends Event
case class FAIL(name: String, nr: Int) extends Event
```

The class `Event` is defined as abstract, meaning that it has to be subclassed. Each kind of event is defined as a subclass of class `Event`. Each event class is furthermore defined as a **case** class, which enables pattern matching over members of the type (this we be illustrated below). Each subclass in this scenario is parameterized with data (the constructor parameters), which must be provided when creating an object of the class. Note that in SCALA constructor parameters can be provided in the class definition without having to define an explicit constructor inside the class as in JAVA. All of the events here have two parameters: a command name of type `String` and a command number of type `Int`. Success and fail events have the command number corresponding to the command they stem from. With the above definitions, the following is an example of a trace of four events:

```
val trace: List[Event] = List(
  COMMAND("STOP_DRIVING", 1), SUCCESS("STOP_DRIVING", 1),
  COMMAND("TAKE_PICTURE", 2), FAIL("TAKE_PICTURE", 2))
```

The **val** keyword introduces a constant, in this case `trace` of type `List[Event]`, defined as the list returned by the list constructor call: `List(`$event_1$, $event_2$,...`)`. Each element in the list is an event, an object of one of the event classes. For example, the first list element `COMMAND("STOP_DRIVING", 1)` is an object of class `COMMAND`. Due to the fact that the event classes are defined as **case** classes, objects can be conveniently created without use of the **new** keyword. Such a trace can for example be constructed by parsing a log file produced by the rover software. Note, however, that events can be processed one by one as well, they do not need to come as part of a pre-computed trace.

2.1 The DSL and a First Example

The main two classes are `Monitor`, offering functions for writing properties, and `Formula`, representing the type of temporal formulas used to define properties. A class `FactOps`

[3] This example is inspired from the Mars Science Laboratory (MSL) rover mission.

offers additional functions on recorded facts (to model past time temporal logic). The API interfaces of these three classes are shown in code listings 1.2 and 1.3 respectively. These, and other listings will be referred to using references of the form: $\langle$*listing-id*:*line-number*$\rangle$ (one line), or: $\langle$*listing-id*:*line-number*$_1$-*line-number*$_2\rangle$ (a range of lines).

Listing 1.2. Class Monitor

```
1  class Monitor[Event] {
2    def property(name: Symbol)(formula: Formula): Unit
3    def invariant(name: Symbol)(block: Block): Unit
4    def monitor(monitors: Monitor[Event]*): Unit
5    def verify(event: Event): Unit
6    def end(): Unit
7    def verify(trace: List[Event]): Unit
8    def finish(): Unit
9    def getMonitorResult: MonitorResult[Event]
10
11   // state logic:
12   type Block = PartialFunction[Event, Formula]
13   def always(block: Block): Formula
14   def state(block: Block): Formula
15   def hot(block: Block): Formula
16   def step(block: Block): Formula
17   def strong(block: Block): Formula
18   def weak(block: Block): Formula
19   def error(message: String): Formula
20   def error: Formula
21   def ok(message: String): Formula
22   def ok: Formula
23
24   // future time temporal logic:
25   def matches(predicate: PartialFunction[Event, Boolean]): Formula
26   def not(formula: Formula): Formula
27   def globally(formula: Formula): Formula
28   def eventually(formula: Formula): Formula
29   def strongnext(formula: Formula): Formula
30   def within(time: Int)(formula: Formula): Formula
31
32   // past time temporal logic:
33   abstract class Fact
34   implicit def convFact2FactOps(fact: Fact): FactOps
35   def factExists(pred: PartialFunction[Fact,Boolean]): Boolean
36
37   // implicit conversions to Formula:
38   implicit def convEvent2Formula(event: Event): Formula
39   implicit def convBoolean2Formula(cond: Boolean): Formula
40   implicit def convUnitToFormula(unit: Unit): Formula
41 }
```

Listing 1.3. Classes Formula and FactOps

```
abstract class Formula {
  // propositional and future time temporal logic:
  def and(that: Formula): Formula
  def or(that: Formula): Formula
  def implies(that: Formula): Formula
  def until(that: Formula): Formula
  def unless(that: Formula): Formula

 // sequential, causal and hierarchical composition:
  def then(that: Formula): Formula
  def causes(that: Formula): Formula
  def except(block: Block): Formula
}

class FactOps(fact: Fact) {
  def + : Unit
  def - : Unit
  def ? : Boolean
  def ~ : Boolean
}
```

The public functions in each class are here represented by their signatures, each introduced with the **def** keyword (the associated bodies are not shown here). Most of these functions will be explained in the following. The class `Monitor` is parameterized with the event type, which must be provided at instantiation time. A user-defined monitor representing one or more trace properties must extend class `Monitor` to get access to the functions defined therein. Consider as an example the following requirements: R_1*: "Whenever a command is issued, it should eventually succeed with no failure occurring before then"*, and R_2*: "A command must not succeed more than once"*. These requirements can be formulated as the TRACECONTRACT monitor in Listing 1.4. The monitor is defined as a class named `CommandRequirements`, which extends the class `Monitor`. In SCALA, the body of a class can contain statements (in addition to definitions), which will get executed when an object is constructed (the body of the class works as the constructor). Each of the two requirements is defined by a call of the function `property` $\langle$1.2:2$\rangle$ from the `Monitor` API. It is a curried function, which as first argument takes the name of the property, and as second argument takes the formula to be checked. The function returns no value of importance (return type is `Unit`), but has as side effect to add the formula to the list of formulas being checked.

Consider the formalization of the first requirement R_1. The first argument to the `property` function is the name of the property, in this case the symbol `'R1` of type `Symbol`. SCALA's `Symbol` type contains quoted names, which are convenient to type instead of strings, such as `"R1"`. The second argument to the `property` function is a formula $\langle$1.4:3-9$\rangle$ of the form: `always{...}`. The formula is the result of a call of the `always` function $\langle$1.2:13$\rangle$, which takes as argument a partial function from events to formulas, called a *block* $\langle$1.2:12$\rangle$. A partial function `f` of type `PartialFunction[`A`,`B`]` is associated with a function `isDefinedAt(x: A):Boolean` where `f.definedAt(v)` returns true if and only if the partial function `f` is defined at `v`. A partial function is typically defined with a sequence of **case** statements, each defining a subset of A for which it is defined. In the above case, the argument to the `always` function is the partial function defined only on `COMMAND` objects (there is only one **case** statement $\langle 1.4{:}4\rangle$):

```
{case COMMAND(name, number) => hot { ... }}
```

When the function is applied to a value v, the value is matched against the patterns in the **case** statements, in a left to right manner, until a match occurs (an exception is thrown in case a match does not occur). In the above example, the value v is matched against the pattern `COMMAND(name, number)`. In case v is a command the match succeeds, the identifiers `name` and `number` are bound to the actual corresponding values in v, and the result is the value of the expression to the right of the `=>` symbol. It is the fact that the event classes `COMMAND`, `SUCCESS` and `FAIL` are defined as **case** classes (Listing 1.1) that allows us to perform pattern matching as above. The intuition is that the `always` function creates a state (a kind of formula), in which the monitor will wait until an event arrives for which the partial function is defined. When this happens, the partial function is applied to obtain a new formula, namely the right hand side of `=>`, in this case a new state H produced by the `hot` function $\langle 1.2{:}5\rangle$. The net result is the conjunction of the original `always{...}` formula and this new state H: `always{...}` $\wedge$ H - to reflect the fact that we will keep checking the body of the `always` function.

Listing 1.4. Formalization of Requirements R_1 and R_2

```
 1  class CommandRequirements extends Monitor[Event] {
 2    property('R1) {
 3      always {
 4        case COMMAND(name, number) =>
 5          hot {
 6            case FAIL(`name`, `number`) => error
 7            case SUCCESS(`name`, `number`) => ok
 8          }
 9      }
10    }
11
12    property('R2) {
13      always {
14        case SUCCESS(_, number) =>
15          state {
16            case SUCCESS(_, `number`) => error
17          }
18      }
19    }
20  }
```

The `hot` function $\langle 1.2{:}15\rangle$ similarly takes a partial function (*block*) as argument. As before, a hot state will remain waiting until an event arrives for which the partial function is defined. However, when such an event arrives, the net result is the value of the body – that is, there is no repetition as in the case of `always`. The `hot` state is also signified by causing an error in case it has not been left before the end of the trace. The partial function occurring as argument to the `hot` function ⟨1.4:6-7⟩ contains patterns which contain quoted variable names: `` `name` `` and `` `number` ``. The meaning of such patterns is that the incoming value must equal the value of these variables, instead of being bound to them. In this example, the monitor is waiting for failure or success of the command previously observed (same name and number). Finally, the formulas `error` ⟨1.2:20⟩ and `ok` ⟨1.2:22⟩ are special formulas, essentially representing `False` and `True` respectively. These functions also exist in overloaded forms taking a message as argument, ⟨1.2:19⟩ and ⟨1.2:21⟩, which is printed to standard out.

The property R_2 ⟨1.4:12-19⟩ states that a success with a certain number should never be followed by another success with the same number. The underscore ('_') is the wildcard pattern that always matches, and is here used to model that the command name is not of importance to this requirement. The `state` function ⟨1.2:14⟩ takes, as before, a partial function (*block*) as argument, and creates a state where the monitor will wait until an event arrives for which the partial function is defined, in which case an error is emitted in this example. In contrast to a hot state, however, no error is issued if a monitor remains in such a state at the end of the trace. `state` states are hence used to model *safety properties*, whereas `hot` states are used to model *liveness properties* (see [6] for a discussion of safety and liveness properties on finite traces).

2.2 State Machines

The API contains other kinds of states, produced by functions that take partial functions as arguments, such as `step` ⟨1.2:16⟩, `strong` ⟨1.2:17⟩, and `weak` ⟨1.2:18⟩ states. A `step` state evaluates to true if it does not trigger in the next step (this corresponds to ignoring this branch). A `strong` state evaluates to false if it does not trigger in the next step (some event *must* happen in the next step). A `weak` state, like a `strong` state, evaluates to false if it does not trigger in the next step, provided there is a next step.

In the example shown in Listing 1.4, properties have a flavor of temporal logic in the sense that intermediate states are not explicitly named. For example, in property `'R1`, the right hand side of the transition '**case** `COMMAND(name, number) => hot{...}`' is a hot state that is not explicitly named, corresponding to a use of the diamond (eventually) operator $\diamond$ in temporal logic. TRACECONTRACT also, however, naturally supports naming of states using SCALA's already built-in function concept, and consequently supports definition of state machines. The two styles (named states and inlined states as in Listing 1.4) can furthermore be mixed freely, which is the main characteristic of the TRACECONTRACT DSL. Consider the requirement R_3*: "Consecutive command numbers should increase by exactly 1, and a command (name) should not be re-issued with a new number until a success has occurred"*. In addition, let's collect the names of the commands issued and store them in a set for later printing. The property is presented in Listing 1.5, including the definition of two functions (`increaseCmdNumber` and `holdCmd`), each representing a parameterized named state. Instead of a call of the form: `property(`*name*`){always{`*block*`}}`, here we use the `invariant` function ⟨1.2:3⟩, giving rise to the abbreviated form: `invariant(`*name*`){`*block*`}`. That is:

```
def invariant(name: Symbol)(block: Block) = property(name){always{block}}
```

Listing 1.5. A State Machine

```
1  var commands: Set[String] = Set()
2
3  invariant('R3) {
4    case COMMAND(name, number) =>
5      commands += name
6      increaseCmdNumber(number) and holdCmd(name, number)
7  }
8
9  def increaseCmdNumber(number: Int) =
10   state {
11     case COMMAND(_, number2) => number2 == number+1
12   }
13
14 def holdCmd(name: String, number: Int) =
15   state {
16     case COMMAND(`name`, number2) if number2 != number => error
17     case SUCCESS(`name`, `number`) => ok
18   }
```

The property illustrates a number of features. Line ⟨1.5:1⟩ declares the monitor local updatable variable `commands` (keyword **var**) of type `Set[String]`, initialized to the empty set. This variable is updated in line ⟨1.5:5⟩ by adding the command name to the set. The variable update is followed in line ⟨1.5:6⟩ by a formula, which is the conjunction of two states. TRACECONTRACT allows conjunction as well as disjunction of states corresponding to AND/OR automata. These two lines illustrate how side-effects elegantly can be combined with logic. The two states are themselves the result of applying the two functions `increaseCmdNumber` and `holdCmd`, defined after the property. Line ⟨1.5:11⟩ illustrates how a Boolean expression (`number2 == number+1`) appears as a formula (it is lifted to a formula by an implicit function ⟨1.2:39⟩). Line ⟨1.5:16⟩ illustrates a conditional transition: the transition is only taken if the pattern ``COMMAND(`name`,number2)`` matches, and the expression `number2 != number` evaluates to true.

2.3 Future Time Temporal Logic

TRACECONTRACT offers, as an alternative, a set of functions supporting writing properties in Linear Temporal Logic (LTL). This includes propositional logic ⟨1.2:26⟩, ⟨1.3:3-5⟩, and temporal operators ⟨1.2:27-29⟩, ⟨1.3:6-7⟩. As an example, the requirement R_1 from above can alternatively be stated as in Listing 1.6, in a notation similar to LTL.

Listing 1.6. An LTL Formula

```
invariant('R4) {
  case COMMAND(name, number) =>
    not(FAIL(name, number)) until SUCCESS(name, number)
}
```

The right hand side of the transition is an LTL formula constructed as follows. First, the events `FAIL(name, number)` and `SUCCESS(name, number)` are each converted to a formula via the implicit conversion function `convEvent2Formula` ⟨1.2:38⟩. Generally, the implicit functions ⟨1.2:38-40⟩ automatically convert values of the argument type into values of the result type. Whenever a SCALA expression fails to type check, the SCALA compiler will consult the implicit functions in scope and determine whether the application of a such will make the expression type check, and in this case the compiler will insert an application of the function (there can be no more than one such implicit conversion function, otherwise the SCALA compiler will complain). These functions allow us to write events, Boolean expressions, and code blocks returning `Unit` as formulas. Second, the formula obtained from the `FAIL(name, number)` event is negated with the `not` function ⟨1.2:26⟩, resulting in a new formula. Listing 1.3 shows the functions callable on formulas, including the `until` function ⟨1.3:6⟩. SCALA permits to write calls of functions on objects without dot-notation and without parentheses around arguments (as required in JAVA). That is, given an object o of a class defining a function m, instead of: $o.m(a)$, we are allowed to write: $o\ m\ a$. This technique is used to write the above LTL formula composing two formulas with the infix `until` operator. The formula is equivalent to: `not(FAIL(name, number)).until(SUCCESS(name, number))`.

As we can see, we have here mixed pattern matching with LTL. In support for handling pattern matching, the API furthermore offers the `matches` function ⟨1.2:25⟩, which returns a formula that will be true or false on an event, depending on whether the partial function argument (a predicate) to `matches` is defined, and furthermore returns true on the event. Note that in this case no binding of values takes place, it is purely a predicate. Amongst other combinators we can mention:- Sequential composition – f_1 `then` f_2 ⟨1.3:10⟩: evaluates f_1 until (and if) it becomes true whereupon f_2 is evaluated; Cause and effect – f_1 `causes` f_2 ⟨1.3:11⟩: whenever f_1 evaluates to true, f_2 is evaluated (similar to message sequence diagrams); Bounds– f `except{`*block*`}` ⟨1.3:12⟩: f is evaluated unless the partial function *block* becomes defined for an incoming event, in which case it is applied and its result becomes the new formula. Several other forms of formula are offered, including formulas counting events, for example that some event must happen within n steps.

2.4 Past Time Temporal Logic

Consider the requirement: R_5*: "A failure should only occur if a command (same name and number) has been observed in the past, and no success has been observed so far since then."*. This requirement expresses a past time property. TRACECONTRACT offers a set of constructs for writing past time properties. The general idea is to support recording of facts in a database, which can then be queried later, i.e. in the future. The API provides an abstract class `Fact` ⟨1.2:33⟩, which the user can extend in order to define facts. An implicit function ⟨1.2:34⟩ converts facts into objects of class `FactOps` ⟨1.3:15⟩, which offers a collection of functions on facts (applied using post-fix notation): a function for adding a fact to the database (`+`), a function for deleting a fact (`-`), and functions for querying whether a fact is in the database (`?`) or not (`~`). There is also a function `factExists` ⟨1.2:35⟩ for checking whether a fact exists that satisfies a predicate. Requirement R_5 can be formulated as in Listing 1.7.

Listing 1.7. Reasoning About the Past

```
1 case class Commanded(name: String, number: Int) extends Fact
2
3 invariant('R5) {
4   case COMMAND(name, number) => Commanded(name, number) +
5   case SUCCESS(name, number) => Commanded(name, number) -
6   case FAIL(name, number) if Commanded(name, number) ~ => error
7 }
```

A class `Commanded` is defined ⟨1.7:1⟩ extending class `Fact`. An object `Commanded(`n`,`x`)` of this class is meant to represent the fact that a command with name n and number x has been observed. By defining it as a **case** class, objects can be conveniently created without using the **new** keyword. The property then updates the database in the first two transitions ⟨1.7:4-5⟩, by adding respectively deleting a fact, and in the third transition ⟨1.7:6⟩ by testing for the absence of the fact (absence is an error). This specification style is very close to the way such past time properties are specified in RULER and LOGSCOPE. Indeed, [7] presents a translation scheme from future and past time LTL to RULER, the past time part of which can be fully mimicked to obtain a formal translation of past time LTL into TRACECONTRACT. Essentially, there is one key difference between the two. In RULER, memory is encoded by rules that are not persistent, i.e. they exist for the next moment only, whereas, here for TRACECONTRACT, facts are persistent by default and must be forcibly removed. Theoretical results about the separation of any LTL formula into pure past, present and pure future parts then enables us to claim, similar to our result for RULER, that any temporal logic property can be embedded/encoded in TRACECONTRACT.

We observe, however, that it is possible, as an alternative to the above, to declare variables in a monitor that maintains such a database and one may then find, because of SCALA's convenient syntax, the incurred effort is not too burdensome.

2.5 Using Monitors

Properties can be written in different monitors and composed in a hierarchical manner by calls of the `monitor(monitors: Monitor[Event]*):Unit` function ⟨1.2:4⟩. This is a function with a variable length argument list, indicated by `'*'`, taking zero or more monitors as arguments. The example in Listing 1.8 illustrates how two sets of requirements are composed into a new monitor `AllRequirements` ⟨1.8:1-3⟩. This class is then instantiated to an object ⟨1.8:7⟩, upon which the `verify` function is called ⟨1.8:9⟩. In this case a trace is read from a log file and the `verify` function ⟨1.2:7⟩ that takes a trace as argument is called. This function will check the events in the trace against the provided properties. In some scenarios, events may be provided in a step-wise manner, for example during online monitoring of a running system, or if the log file is too large to be represented as a SCALA list. In such cases the alternative `verify(event: Event)` function ⟨1.2:5⟩ can be called on each incoming event. Such a sequence of calls has to be ended with a call of the `end` function ⟨1.2:6⟩. The user can in the monitor override the `finish` function ⟨1.2:8⟩ (for example to compute and print some statistics), which will be called when `end` is called.

Listing 1.8. Using a Monitor

```
class AllRequirements extends Monitor[Event] {
  monitor(new CommandRequirements, new RadioRequirements)
}

object TraceAnalysis {
  def main(args: Array[String]) {
    val monitor = new AllRequirements
    val trace = readLog()
    monitor.verify(trace)
  }
}
```

3 Implementation

In this section we shall briefly outline how the combinators described above have been implemented. We shall leave out non-essential details.

3.1 Formulas and Linear Temporal Logic

Requirements to be monitored are expressed as formulas. Formulas are objects of subclasses of the abstract class in Listing 1.9.

Listing 1.9. Class Formula

```
abstract class Formula {
  def apply(event: Event): Formula
  def reduce(): Formula = this
  def and(that: Formula): Formula = And(this, that).reduce()
  def until(that: Formula): Formula = Until(this, that).reduce()
  ...
}
```

Each kind of formula is represented by a specific subclass of this class. A number of functions are defined on all formulas, four of which are shown here. The `apply` function takes an event and returns a new formula, either unchanged in case the event is not relevant, or a changed to reflect the impact of the event. The `apply` function is special in SCALA in that for a given formula `f`, it allows us to write `f(e)` instead of `f.apply(e)`. The function is defined as abstract and is overridden by the different subclasses of `Formula`.

The `apply function` is invoked as follows. A monitor consists in principle of a collection of formulas to be monitored. The `verify` function ⟨1.2:7⟩ applied to a trace will traverse the trace, and will for each event *e* call `verify(`*e*`)` ⟨1.2:5⟩, which in turn will apply each formula f in the monitor to the event: $f(e)$, resulting in either True, False, an unchanged formula, or a changed formula different from True and False. At the end of the trace (when the `end` function ⟨1.2:6⟩ is called) all formulas are evaluated to either true (if they represent safety properties) or false (if they represent liveness properties).

The `Formula` class furthermore contains all infix operators on formulas, including Boolean logic operators, such as `and`, as well as temporal operators such as `until`. For example, given two formulas f_1 and f_2, the function `and` allows us to write: f_1 `and` f_2 (instead of the more classical also allowed: f_1`.and(`f_2`)`). The result is an object of class `And`, which is one of the many subclasses of class `Formula`, see Listing 1.10.

When composing Boolean logic expressions, it is necessary to simplify them. For example, for a formula f: $\mathit{true} \wedge f$ can be reduced to: f. The `reduce` function will perform this rewriting according to the classical Boolean axioms for each formula resulting during monitoring. By default the function is defined to leave the formula unchanged, but may be overridden in subclasses of `Formula`.

The atomic formulas are `True`, `False`, and `Now(`e`)`, for some event e. The latter formula is true if the current event is equal to e. These atomic combinators are defined as subobjects/classes of `Formula`. A term such as `And(`f_1`,`f_2`)` is evaluated by evaluating its subformulas, and subsequently calling `reduce` to perform Boolean logic reduction, as shown in Listing 1.10.

Listing 1.10. Class And

```scala
case class And(formula1: Formula, formula2: Formula) extends Formula {
  override def apply(event: Event): Formula =
    And(formula1(event), formula2(event)).reduce()

  override def reduce(): Formula = {
    (formula1, formula2) match {
      case (False, _) => False
      case (_, False) => False
      case (True, _) => formula2
      case (_, True) => formula1
      case (f1, f2) if f1 == f2 => f1
      case _ => this
    }
  }
}
```

Here `reduce` is defined with a **match** statement: the tuple `(formula1, formula2)` is matched against the patterns `(False, _)`, `(_, False)`, etc., until there is a match, and the formula on the right hand side of the corresponding => symbol is returned. The formulas corresponding to the classical LTL operators $\Box$ (globally) and $\Diamond$ (eventually) are defined using the classical rewrite rules '$\Box p = p \wedge \bigcirc\Box p$' and '$\Diamond p = p \vee \bigcirc\Diamond p$', as shown in Listing 1.11.

Listing 1.11. Classes Globally and Eventually

```scala
case class Globally(formula: Formula) extends Formula {
  override def apply(event: Event): Formula =
    And(formula(event), this).reduce()
}
```

```

case class Eventually(formula: Formula) extends Formula {
  override def apply(event: Event): Formula =
    Or(formula(event), this).reduce()
}
```

We have mentioned that events, Booleans and the unit value are automatically transformed to formulas. This is achieved through the definitions in Listing 1.12. The conversion from the unit value to `True` allows us to write a block of code in the place of a formula, which can be useful when writing state machines.

Listing 1.12. Implicit Functions

```
implicit def convEvent2Formula(event: Event): Formula = Now(event)
implicit def convBoolean2Formula(cond: Boolean): Formula =
  if (cond) True else False
implicit def convUnitToFormula(unit: Unit): Formula = True
```

3.2 State-oriented Constructs

The most common specification style in TRACECONTRACT is to use (anonymous or named) states, The formula classes of some of these are shown in Listing 1.13. Common for all states is that they consist of a block ⟨1.2:12⟩, which is a partial function from events to formulas. States differ in how they evaluate when the partial function is not defined for an event. That is, whether they become `True`, `False`, or stay unchanged (**this**).

Listing 1.13. States

```
case class State(block: Block) extends Formula {
  override def apply(event: Event): Formula =
    if (block.isDefinedAt(event)) block(event) else this
}

case class Hot(block: Block) extends Formula {
  override def apply(event: Event): Formula =
    if (block.isDefinedAt(event)) block(event) else this
}

case class Step(block: Block) extends Formula {
  override def apply(event: Event): Formula =
    if (block.isDefinedAt(event)) block(event) else True
}

case class Strong(block: Block) extends Formula {
  override def apply(event: Event): Formula =
    if (block.isDefinedAt(event)) block(event) else False
}
```

States, and other formulas, also differ in the way they evaluate at the end of the trace. Some formulas will evaluate to true (representing safety properties: nothing unexpected happened), while others will evaluate to false (representing liveness properties: something expected did not happen). For example, for any formula f, at the end the formula `Globally(`f`)` evaluates to true whereas `Eventually(`f`)` evaluates to false. Likewise, for any block b, `State(`b`)` evaluates to true whereas `Hot(`b`)` evaluates to false.

3.3 Properties, Formulas, and Error Traces

A monitor technically contains a collection of properties. A property is a named formula. A property also maintains an error trace candidate for the formula. Whenever the formula changes due to a new incoming event, that event is recorded in the error trace. This way an error trace for a formula reflects only those events that are important to the evolution of the formula. If the formula at some point is violated, that error trace can be printed to the user. However, formulas of the form `always{`f`}`, for some formula f, are treated differently than other formulas at the top level in order to provide the user with informative error traces. That is, in case such a formula occurs and f changes to f' different from True or False, then a new property is created specifically for the new f', with a new error trace initialized to contain the event that caused the formula change. The original `always{`f`}` continues to be monitored as well, reflecting the semantics: $\Box p = p \wedge \bigcirc\Box p$. Consider as an example the formula `always{`**case** `e => eventually(q)}`. Each time `e` matches an incoming event, a new property monitoring `eventually(q)` is created, tracking the error trace only for this particular scenario.

4 Conclusion

TRACECONTRACT is implemented as an API in SCALA, also referred to as a *shallow internal* DSL. Internal since it extends the host language (SCALA) and shallow since it is defined relying heavily on SCALA's already existing language constructs, such as function definitions, partial functions and pattern matching. An immediate consequence is that, in contrast to RULER, TRACECONTRACT is close to an order of magnitude smaller in code size, even though it offers greater functionality and easier adaptability. Previously, we have had many discussions as to how to integrate temporal logic into the RULER and LOGSCOPE systems. The internal DSL surprisingly provides many of these concepts with very little effort. Using SCALA as a host language can, however, potentially in some contexts be considered as a drawback instead of a virtue, depending on who the user is. Flight missions at NASA are for example more often manned with system/hardware engineers than with software engineers. One cannot expect a system engineer to use a programming language such as SCALA. A system engineer is more likely to pick up an external DSL with limited scope and limited potential for introducing programming errors. This dilemma is a subject for further research.

Our approach with TRACECONTRACT in SCALA has led to a very expressive and convenient DSL, but at the cost of analyzability. That is, a piece of specification is a SCALA fragment, and SCALA does not offer enough reflexive capabilities to allow one to analyze such a fragment, unless one interferes with the SCALA compiler. Amongst the things that become difficult is to provide the user with detailed information about the specification, such as visualizing state machines, or showing the progress of monitoring. It might also become a challenge to maximally optimize the implementation.

TRACECONTRACT was initially developed for analysis of log files produced from running software. For this purpose we believe that the solution is very powerful and convenient. The system can, however, also be used for online monitoring. Indeed, online monitoring of SCALA programs is a natural application, and future work includes extending Odersky's design by contract [14] with TRACECONTRACT.

References

1. NASA's LADEE (Lunar Atmosphere and Dust Environment Explorer) mission. http://www.nasa.gov/mission_pages/LADEE/main.
2. The Scala programming language. http://www.scala-lang.org.
3. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittamplan, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA'05*. ACM Press, 2005.
4. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
5. H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal analysis of log files. *Journal of Aerospace Computing, Information, and Communication*, 7(11):365–390, 2010.
6. H. Barringer, K. Havelund, D. Rydeheard, and A. Groce. Rule systems for runtime verification: A short tutorial. In *Proc. of the 9th Int. Workshop on Runtime Verification (RV'09)*, volume 5779 of *LNCS*, pages 1–24. Springer, 2009.
7. H. Barringer, D. E. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from EAGLE to RULER. *J. Log. Comput.*, 20(3):675–706, 2010.
8. F. Chen and G. Roşu. MOP: An efficient and generic runtime verification framework. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, 2007.
9. D. Drusinsky. The temporal rover and the ATG rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.
10. B. Finkbeiner, S. Sankaranarayanan, and H. Sipma. Collecting statistics over runtime executions. *Formal Methods in System Design*, 27(3):253–274, 2005.
11. F. Garillot and B. Werner. Simple types in type theory: Deep and shallow encodings. In *20th Int. Conference on Theorem Proving in Higher Order Logics (TPHOLs'07), Kaiserslautern, Germany.*, volume 4732 of *LNCS*, pages 368–382. Springer, 2007.
12. K. Havelund and G. Rosu. Monitoring programs using rewriting. In *16th ASE conference, San Diego, CA, USA*, pages 135–143, 2001.
13. I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *PDPTA*, pages 279–287. CSREA Press, 1999.
14. M. Odersky. Contracts for Scala. In *Runtime Verification - First Int. Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *LNCS*, pages 51–57. Springer, 2010.
15. V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *Proc. of the 5th Int. Workshop on Runtime Verification (RV'05)*, volume 144(4) of *ENTCS*, pages 109–124. Elsevier, 2006.
16. V. Stolz and F. Huch. Runtime verification of concurrent Haskell programs. In *Proc. of the 4th Int. Workshop on Runtime Verification (RV'04)*, volume 113 of *ENTCS*, pages 201–216. Elsevier, 2005.