Space Telemetry Analysis with PyContract*

Bevin Duckett[1], Klaus Havelund[1], and Luke Stewart[1]

Jet Propulsion Laboratory, California Inst. of Technology, USA

Abstract. PyContract is a Python library for trace analysis, also characterized as an internal DSL (Domain-Specific Language). It combines flavors of state machines and rule-based programming, supporting states that can carry data, thus allowing for monitoring of events that carry data. The fact that it is a Python library offers full expressiveness, and access to a vast amount of libraries, which becomes useful for realistic situations. This is in this paper illustrated by its real life application to data analysis of telemetry logs obtained during testing of NASA's Europa Clipper flight computer. The mission will place a spacecraft in orbit around Jupiter in order to perform a detailed investigation of its moon Europa. The analysis includes not only verifying functional correctness but also, and especially, performance analysis such as execution times and rates of change. This includes generation of data in table format and visualization as graphs. The important message is that runtime verification and data analysis are closely related topics, which can only be addressed with highly expressive specification languages.

1 Introduction

Runtime Verification (RV) is normally seen as a discipline of verifying whether a system/program execution is correct wrt. a given set of properties, yielding a Boolean true/false flavored verdict[1]. It can with this view be seen as a lightweight formal method, where the specification is formal, but where only single executions are checked, in contrast to all executions, or even necessarily many executions. Runtime verification is complementary to test case generation but can be used for formulating test oracles, or it can be applied after deployment of the system in the real world to verify that the system performs as desired during operation. Properties are usually expressed in formal Domain-Specific Languages (DSLs) of temporal nature, such as e.g. various forms of temporal logic, regular expressions, state machines, grammars, rule-based systems, and stream processing formalisms. Runtime verification can be applied *online*, monitoring a system as it executes, or it can be applied *offline*, analysing a log produced by a previous run of a system. In this paper we shall study offline RV, combining classical

* The research performed was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

[1] Some RV theories operate with extensions of the Boolean domain with a small finite set of additional values, e.g. [6].

Boolean verdict RV with data analysis, where the focus is on producing data from logs.

A classical distinction amongst DSLs is that of external versus internal DSLs [17]. An *external* DSL is a "small language" with its own grammar and parser. An *internal* DSL (sometimes referred to as an *embedded* DSL) is a library in a general purpose programming language. Numerous external RV DSLs have been developed over time, including [23, 2, 12, 4, 9, 25, 5, 22, 15, 1]. Internal DSLs are usually again grouped into deep and shallow [18]. In a *deep* internal DSL, data structures in the host language are used to represent DSL constructs in an explicit manner, e.g., as an AST (Abstract Syntax Tree), which can then be processed by writing either an interpreter or a compiler for execution. Some examples are [19, 30]. A *shallow* internal DSL includes the constructs of the host language as part of the DSL, using the host language's native runtime system to execute them. Examples of shallow internal DSLs in Scala include [3, 20, 21, 24].

PyContract [10] is a very expressive internal shallow DSL for runtime verification. In this work we present its application to data analysis of telemetry from NASA's Europa Clipper mission [16] flight computer during real life testing of its performance. We present sketches of five such monitors. The purpose is not only to check functional correctness of temporal properties, but also to analyze and visualize non-functional properties, such as performance wrt. time and data volumes. The possible advantages of using PyContract for such data analysis has been previously suggested in [11]. The mentioned advantages include the fact that Python is already highly popular in the field of data analysis, and e.g. used extensively within NASA's Jet Propulsion Laboratory (JPL) for telemetry analysis on ground, examining data coming from spacecraft and rovers. An internal DSL is "just" another library in a familiar language. It allows to use favorite development tools (such as IDEs) and other libraries for the host language. It is expressive, and implementation and maintenance of the DSL is easier. A disadvantage of shallow internal DSLs (when compared to external DSLs and deep internal DSLs) is lack of *analyzability*. However, Python supports powerful meta-programming features allowing a program to inspect its own AST. We use this for visualizing specifications.

Although only briefly touched upon in the paper, our work includes a web-based interface to the use of PyContract, programmed using the Dash visualization library [13]. It provides a unified convenient framework for requesting analyses to be performed as well as for visualizing and tabulating results.

The paper is organized as follows. Section 2 introduces the PyContract library. Section 3 describes how it is applied to the five different data analysis problems, each resulting in a PyContract monitor. Section 4 concludes the paper.

2 The PyContract Core Library

PyContract allows to specify first-order temporal properties over a trace of events. A temporal property relates events occurring at different points in the

trace. The first-order capability allows to also relate the data occurring in events across different points in the trace, turning the logic very expressive. The fact that PyContract is a Python library furthermore augments the expressiveness, allowing to combine temporal properties and general purpose programming, making it Turing complete. PyContract is inspired by rule-based programming [4, 21] in that the memory of a monitor is a set of facts, where a fact in its basic form is a named data record. However, facts, like states in state machines, can have transitions which, upon triggering, can generate other facts, while removing the fact who's transition is taken. In the following we shall demonstrate how this looks like. PyContract is inspired by the Scala DSL Daut [20, 14] and is developed for Python 3.10 that supports pattern matching [29]. PyContract is available under the Apache 2.0 open-source license at [28].

The general approach is to define a monitor as a sub-class of the `Monitor` class, create an instance of it, and then feed it with events, as shown in Figure 1. Events can be fed, one by one, using the `evaluate(event: object)` method. In the case of a finite sequence of observations, for example when examining a log file, a call of the `end()` method tells the monitor that the sequence has ended. Note that `end()` may not be called when monitoring is online, but if it is called, any outstanding obligations that have not been satisfied (expected events that did not occur) will be reported as errors. Events can be provided by the user in different ways. If examining a log file for example, they can be read from any file format such as e.g. CSV, JSON, XSML, etc. This would in the example in Figure 1 take place in line 7, where we instead of providing the trace explicitly, as done here, would read it from a file.

```
1  import pycontract as pc
2
3  class M(pc.Monitor):
4      ... body of monitor ...
5
6  monitor = M()
7  trace = [event_1, event_2, ..., event_n]
8  for event in trace:
9    monitor.evaluate(event)
10 monitor.end()
```

Fig. 1. General approach for defining and using a PyContract monitor.

As an example we will define a monitor for verifying command execution on board the spacecraft. The example yields a Boolean verdict, in the tradition of classical RV. Commands are submitted to the spacecraft, and on board dispatched, followed hopefully by a completion. Commands have a name and each command dispatch has in addition a number. Events can in PyContract be any data object. We shall in this paper focus on events represented as dictio-

naries: mapping from fields to values. E.g. the dispatch of a command may be represented by the event:

$$\{\text{“}\texttt{name}\text{”} : \text{“}\texttt{dispatch}\text{”}, \text{“}\texttt{cmd}\text{”} : \text{“}\texttt{TURN}\text{”}, \text{“}\texttt{nr}\text{”} : 3, \text{”}\texttt{time}\text{”} : 382649\}$$

We shall verify the following property:

> **Commands***: The dispatch of a command, with a number, must be followed by a completion within 3 seconds, and no failure of that command dispatch must be observed in between. In addition, a dispatch number can only be completed once (no double execution).*

Note that the “*within 3 seconds*” constraint can have two interpretations: either that a failure is reported exactly after 3 seconds without having seen a completion, or that a failure is reported when observing an event occurring after 3 seconds without having seen a completion within 3 seconds. The former interpretation requires an internal clock in the monitor, whereas the latter interpretation can rely on the time stamps carried by events. We adopt the latter interpretation, which is suitable for log analysis. For online runtime verification, however, the former interpretation would be more appropriate.

The property is implemented as the monitor in Figure 2. First we import the PyContract module (line 1). The monitor is defined as a class extending the `Monitor` class (line 3). The body of the monitor defines a `transition` function (lines 4-7), and two states: `DoComplete` (lines 10-20) and `Executed` (lines 23-29). The outer `transition` function (lines 4-7) processes all events submitted to the monitor. It takes an event as argument and matches it against possible patterns, using the pattern matching features provided in Python from version 3.10 [29]. In this case just one pattern matches if the name of the command is `"dispatch"`. If so it binds the command id, number, and time to the variables `c`, `n`, and `t` respectively, and returns a new state: `DoComplete` with these bindings as arguments. This state is now added to the memory of the monitor. The actual type of the `transition` function is:

```
def transition(self, event: Event) ->
  Optional[State | List[State]]
```

where `Event` is the type of events (dictionaries in this case). It returns either `None` (corresponding to no match), a state, or a list of states. We leave out the types in the remaining transition function definitions.

The `DoComplete` state extends `HotState`, meaning that it must eventually be removed, otherwise an error is reported when the `end()` method is called at the end of the trace. It will e.g. be removed when the command it monitors completes. The state is parameterized with a command id, a dispatch number, and the time of dispatch. The body defines a transition function applicable when the state is active, which offers three options for processing an incoming event (if none match the state remains in the monitor memory). The first case

matches if the command (with the same id and number) fails[2]. In this case an error is reported. The second case matches if any event is observed with a time stamp more than 3 seconds from the dispatch time. This also results in an error being reported. The third case matches if the command completes, in which case an `Executed` state, parameterized with the dispatch number, is returned, and recorded in the monitor memory (while the `DoComplete` state is removed). The `Executed` state itself is just a `State`, meaning that it is ok to terminate in this state. It monitors that the dispatch number does not complete again.

We mentioned above that the outermost transition function (lines 4-7) is applied to all events submitted to the monitor. Behind the scenes it is translated to an initial so-called `AlwaysState`, as shown in Figure 3 (lines 2-7). An `AlwaysState` state is always active. The former style is, however, more convenient to write.

PyContract offers other features, such as allowing to return a list of states from a transition, next-states (failing if no transition cases match an event), querying the fact memory (used for expressing past time properties), grouping of monitors, and user-defined indexing (slicing) to optimize monitoring, similar to what is supported in RV systems such as MOP [25] and QEA [30]. In addition one can of course add any Python code to be executed in transition actions, and use general Python expressions as transition conditions. PyContract was evaluated against other systems in [10], performing reasonably by processing 4 million events in under 100 seconds.

PyContract visualizes a monitor using PlantUML [27] by first analyzing the AST of the monitor (using Python's meta-programming capabilities) and then generating PlantUML text. Figure 4 is such a visualization of the monitor in Figure 2. Green states (the initial state) are always active, and safe to terminate in. Bright yellow states, the `DoComplete` state, indicate danger: they must be left eventually. Faded yellow states, the `Executed` state, are safe to terminate in as well. Finally red states are error states. Transitions out of a state are numbered to indicate the order of evaluation caused by the semantics of Python's **`match`**-statements.

3 Data Analysis Scripts

In this section we present five different monitors using the PyContract library for performing various forms of combined property checking and data analysis. We show only essential code fragments that provide the general idea. The scripts offer user options for different behaviours, which we largely ignore in this presentation. A spacecraft reports telemetry to ground as individual messages. There are three general types of spacecraft data sent to ground [7]:

[2] In Python's pattern matching, dotted names, such as `self.cmd`, must match the incoming value, whereas non-dotted names, such as `c`, are binding the incoming value.

```
import pycontract as pc

class Commands(pc.Monitor):
  def transition(self, event):
    match event:
      case {'name':'dispatch','cmd':c,'nr':n,'time': t}:
        return Commands.DoComplete(c, n, t)

  @pc.data
  class DoComplete(pc.HotState):
    cmd: str; nr: int; time: int

    def transition(self, event):
      match event:
        case {'name':'fail','cmd':self.cmd,'nr':self.nr}:
          return pc.error('failed')
        case {'time':t} if t - self.time > 3000:
          return pc.error('time')
        case {'name':'complete','cmd':self.cmd,'nr':self.nr}:
          return Commands.Executed(self.nr)

  @pc.data
  class Executed(pc.State):
    nr: int

    def transition(self, event):
      match event:
        case {'name':'complete','nr':self.nr}:
          return pc.error('double completion')
```

Fig. 2. A monitor for property **Commands**.

- *Time series data* (EHAs[3]) representing onboard measurements of spacecraft state over time. JPL missions generally refer to this type of data as "channelized telemetry" or "channels", with each channel representing a time series of measurements from spacecraft hardware sensors, as well as data reported by software components (e.g. onboard memory states).
- *Event Records* (EVRs) representing single events that occur onboard the spacecraft. Rather than the single data value of a channel record, each EVR record contains a message string, which contains further spacecraft state information embedded in that message.
- *Data Products* (DPs), each containing a range of types of information, depending on the need. There are a wide variety of data products used by projects, including snapshots of state such as memory and data management states.

[3] EHA stands for 'Engineering Housekeeping & Accountability'.

```
class Commands(pc.Monitor):
  @pc.initial
  class Start(pc.AlwaysState):
    def transition(self, event):
      match event:
        case {'name':'dispatch','cmd':c,'nr':n,'time':t}:
          return Commands.DoComplete(c, n, t)

  @pc.data
  class DoComplete(pc.HotState):
    ...

  @pc.data
  class Executed(pc.State):
    ...
```

Fig. 3. Translation of the outermost transition function of the **Commands** monitor in Figure 2 to an AlwaysState containing the transition function.

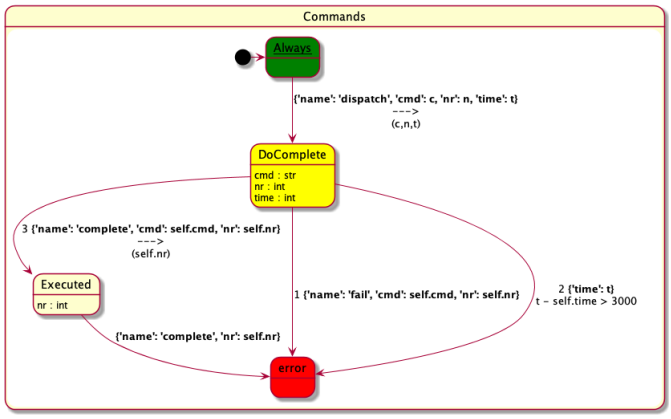

Fig. 4. Visualization of the monitor in Figure 2, generated by PyContract.

In this work we are only concerned here about the first two kinds (EHAs and EVRs). The five monitors will analyze logs containing sequences of such spacecraft data. These include (1) counting of EHAs per 5 seconds, illustrating a very basic monitor not using state machines; (2) reporting of EVRs that occur within a certain time frame after having been reported missing, illustrating a temporal property formulated as state machine with multiple states active, as well as the handling of time in such a state machine; (3) file uplink to the spacecraft, consisting of several events that must occur in order, and where at the end several statistics are computed, illustrating data storage, many states, and hot states; (4) verification that issued commands are followed by expected responses (suc-

cess, failure), and the durations of these (minimal, maximal, average, median), illustrating trace slicing for optimization and modeling of past time properties; and finally (5) measuring rates with which sampled values change, illustrating a more complicated past time property. The examples are non-trivial, and demonstrate the combination of Boolean first-order temporal properties combined with data analysis going beyond Boolean verdicts.

3.1 The Sample Counting Monitor

Our first monitor shows the number of channel values (EHAs) that are received per 5 seconds. Figure 5 shows the first lines of an example CSV file[4]. Each row reports the reading of a channel in a particular software module on board the spacecraft. Specifically, column D contains channel IDs of the form `<module>-<chan>`, consisting of a module name and a channel number. Column J contains time stamps of the form `<year>-<day>T<hour>:<min>:<sec>.<ms>`. To process this we can import and use various Python libraries, in this case `csv` (for reading CSV files), `re` (regular expressions), `datetime` (handling of time stamps), `statistics` (for statistics), and the substantial data analysis and visualization libraries `pandas` [26] and `dash` [13], illustrating why an internal programming oriented monitoring library is useful.

	A	B	C	D	E	F	G	H	I	J	K
1	...	...	...	M1-1	...	...	...	...	...	2022-277T01:59:28.526153	...
2	...	...	...	M2-2	...	...	...	...	...	2022-277T01:59:28.526153	...
3	...	...	...	M3-3	...	...	...	...	...	2022-277T01:59:29.027038	...
4	...	...	...	M3-4	...	...	...	...	...	2022-277T01:59:29.027038	...
5	...	...	...	M3-5	...	...	...	...	...	2022-277T01:59:29.027038	...
6	...	...	...	M3-6	...	...	...	...	...	2022-277T01:59:29.027038	...
7	...	...	...	M4-7	...	...	...	...	...	2022-277T01:59:30.151107	...
8	...	...	...	M4-8	...	...	...	...	...	2022-277T01:59:30.151107	...
9	...	...	...	M4-9	...	...	...	...	...	2022-277T01:59:30.151107	...
10	...	...	...	M4-10	...	...	...	...	...	2022-277T01:59:30.151107	...
11	...	...	...	M4-11	...	...	...	...	...	2022-277T01:59:30.151107	...
12	...	...	...	M4-12	...	...	...	...	...	2022-277T01:59:30.151107	...
13	...	...	...	M4-13	...	...	...	...	...	2022-277T01:59:30.151107	...

Fig. 5. An example of a log represented as a CSV file.

Figure 6 shows the type of events (used for all scripts), namely that of dictionaries from CSV column names to values. The function `convert` (line 8-12) takes as argument an event and augments it with additional fields, in this case the module in which the channel is sampled, and the time. This approach of extending events with additional "columns" is used as a general approach to deal with data fields, who's composition needs processing before being referred to in monitors.

Figure 7 illustrates a statistics module that our monitor will instantiate and update. The essence here is that of going beyond Boolean verdict monitoring. The statistics module maintains a list of channel reading counts per 5 seconds,

[4] Data have been left out or renamed to keep sensitive data hidden.

```
  Event = Dict[str, object]

  def stamp_to_datetime(cls, timestamp: str) -> DateTime:
    return datetime.strptime(timestamp, '%Y-%jT%H:%M:%S.%f')

  chan_pat = re.compile(r'(\w+)-(\d+)')

  def convert(self, event: Event) -> Event:
    module = chan_pat.match(event['D'])[1]
    date_time = Time.stamp_to_datetime(event['J'])
    event.update({'Module': module, 'Time': date_time})
    return event
```

Fig. 6. The event type and functions for extending events.

and a mapping from module names to the number of channel readings in that module. Finally the results can be shown textually and in graphs, implemented using Python's `dash` library.

```
class Statistics:
  def __init__(self):
    self.counts: List[int] = []
    self.modules: Dict[str, int] = {}

  def to_text(self): ...
  def to_graph(self): ...
```

Fig. 7. The statistics class.

Finally, our monitor can be programmed as shown in Figure 8. Note that this monitor represents a basic case where no states are needed, only the top level `transition` function. It corresponds to basically just writing a program. We have shown it here to illustrate how also such a monitor can be made to fit into the library's vocabulary, extending the `Monitor` class and defining the `transition` function. The result of running the monitor is statistics about how many channels were read per 5 seconds, an example is visualized in Figure 9, as well as various tables, including e.g. one showing how many readings that were observed per module, see Figure 10.

3.2 The Missed Event Monitor

The second monitor in Figure 11 highlights (as its output) any row that reports a missing EVR (line 4), which then occurs anyway (line 19) with a matching name later within 5 seconds, and without any intervening rows reporting a timeout (line 13), another failure report for the same EVR (line 15), or a success report

```
1  class EHACount(pc.Monitor):
2    def __init__(self):
3      super().__init__()
4      self.previous_time: DateTime = None
5      self.current_time: DateTime = None
6      self.eha_count: int = 0
7      self.statistics = Statistics()
8
9    def transition(self, event):
10     module = event['Module']
11     self.statistics.update_modules(module)
12     time = event['Time']
13     if self.previous_time is None:
14       self.previous_time = time
15     self.current_time = time
16     seconds = diff(self.previous_time, self.current_time)
17     if seconds < 5:
18       self.eha_count += 1
19     else:
20       self.statistics.update_counts(self.eha_count, seconds)
21       self.eha_count = 1
22     self.previous_time = self.current_time
23
24   def end(self):
25     self.statistics.to_text()
26     self.statistics.to_graph()
```

Fig. 8. The channel sample counting monitor.

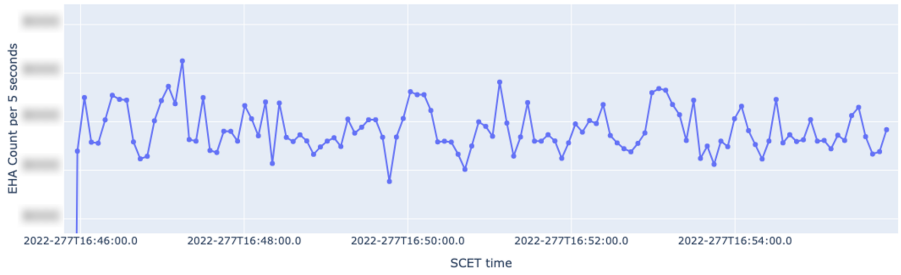

Fig. 9. Graphing of channel counts per 5 seconds.

for that EVR (line 17). This monitor is temporal in nature in that upon detection of a reported missing event (line 4), it creates a new `Watch`-state, parameterized with the EVR name and the time. The `Watch`-state subsequently watches rows relevant for that EVR. Note that if several EVRs are missing a `Watch`-state will

Module	Count
M23	234
M6	235
M11	821
M13	826
M1	1,054
M20	1,989

Fig. 10. Tabulation of channel counts per module.

be created for each. The monitor demonstrates a temporal property combined with reporting, via calls of the `info` method, of events that modify its state.

```
class MissingEVR(pc.Monitor):
  def transition(self, event):
    match event:
      case {Col.Kind:Kind.FAILED,Col.Evr:evr,Col.ERT:ert}:
        return MissingEVR.Watch(evr, ert)

  @data
  class Watch(State):
    evr: str; ert: str

    def transition(self, event):
      match event:
        case {Col.ERT:ert} if diff(self.ert, ert) > 5:
          return pc.info('timed␣out', ...)
        case {Col.Kind:Kind.FAILED,Col.Evr:self.evr}:
          return pc.ok
        case {Col.Kind:Kind.RECEIVED,Col.Evr:self.evr}:
          return pc.info('successfully␣received', ...)
        case {Col.ID:self.evr,Col.TYPE:ty,Col.ERT:ert}
        if ty.startswith('EVR'):
          return pc.info('EVR␣detected', ...)
```

Fig. 11. The missing EVR monitor.

3.3 The File Uplink Monitor

The objective of the next monitor is to report on file uplinks from Earth to spacecraft. A file uplink is recorded in the telemetry as a sequence of EVRs, each providing additional information about the uplink. This information must be gathered and shown, including durations between EVRs, the total duration, the file size, and file size divided by duration, etc. In addition various statistics across file uplinks must be tabulated.

Figure 12 sketches the monitor for this analysis. Upon detecting the start of a file uplink (line 8) a `ReceiveMeta` state is created, that now looks for the next relevant event in the file uplink process. The monitor illustrates a number of points that one normally does not see in temporal specifications. First, we define storage to keep track of statistics across file uplinks (line 4). This resembles the variable state of an extended finite state machine [8]. Second, instead of passing numerous parameters to each state, a single object containing all data for a particular uplink sequence is created and passed as argument (line 9). This is then parameter to each state (e.g. line 13), and can be updated (line 21) before being passed on. Three more states are needed (lines 24-31), all following the same pattern that one is replaced by the next upon a certain event, and where in the final `Finish` state, upon detecting the end of the uplink, statistics is printed out. Note that the `pc.ok` return state (line 18) indicates that monitoring of this particular file uplink is terminated, a `FIT_INFO` event aborts the monitoring of this particular uplink.

3.4 The Command Execution Monitor

The next monitor examines the execution of commands and tabulates their execution time (minimal, maximal, average, median). The monitor also verifies that expected responses (success, failure) follow the dispatch of commands. That is, a dispatched command must be followed by a success or failure, with command id and number (and other parameters) matching (all extracted by regular expressions from data columns). Furthermore, a success or failure of a command that has not been dispatched should cause an error.

The monitor in Figure 13 performs this analysis. The monitor creates a `Succeed` state upon detection of a command dispatch (line 19), which then watches for success or failure of the command. The monitor illustrates a few points. First of all, as previously, we notice the statistics updating (lines 4 and 32). The monitor also shows the definition of the `key` function (lines 6-9), which overrides its definition in the `Monitor` class (where it returns `None`) to return the hash value of an event. This is used for storing `Succeed` states in hash buckets for faster lookup. As mentioned before, this corresponds to slicing as found in systems such as MOP [25] and QEA [30]. A final comment concerns the definition of the `monitored` function (lines 11-14), which is called as a transition condition (line 21), returning true if a `Succeed` state exists in the monitor memory, with appropriate command code and number. It is here used to flag if a command succeeds or fails without a previous dispatch.

```
class Uplink(pc.Monitor):
  def __init__(self):
    super().__init__()
    self.statistics = Statistics()

  def transition(self, event):
    match event:
      case {Col.TYPE:Val.FIT_INFO,Col.Cmd:c,Col.File:f}:
        return Uplink.ReceiveMeta(UplinkInfo(c, f))

  @pc.data
  class ReceiveMeta(pc.HotState):
    info: UplinkInfo

    def transition(self, event):
      match event:
        case {Col.TYPE:Val.FIT_INFO,Col.Cmd:self.info.cmd}:
          return pc.ok
        case {Col.SOURCE:source,Col.ID:Val.RECEIVE_META, ...}
        if source in [Val.FSW_RT, Val.FSW_REC]:
          self.info.source = source
          return Uplink.ReceiveEOF(self.info)

  @pc.data
  class ReceiveEOF(pc.HotState): ...

  @pc.data
  class Succeed(pc.HotState): ...

  @pc.data
  class Finish(pc.HotState): ...
```

Fig. 12. The file uplink monitor.

3.5 The Sample Rate Monitor

The last analysis reports, amongst other things, the rates with which channels change per second, measured for each channel in periods of 15 seconds, called *rate events*. The data can be collected in two modes chosen by the user with an option: (1) across the entire log, or (2) in so-called autopsy windows only. Autopsy windows are 60 second periods where the spacecraft records autopsy information in a buffer, which is then later dumped to a data product and sent to ground, indicated by a *recording off* EVR at the end of the window. During analysis, however, we do not know when a 60 second window begins until we see the *recording off* EVR, complicating the analysis from a temporal point of view. Events overlapping an autopsy window are considered relevant if they terminate within 60 seconds after the *recording off* EVR. The monitor reports in table

```
class CommandDur(pc.CSVMonitor):
  def __init__(self):
    super().__init__()
    self.statistics = Statistics()

  def key(self, event) -> Optional[object]:
    match event:
      case {Col.CmdCode:c,Col.CmdNr:n}:
        return int(n)

  def monitored(self, code: str, nr: str):
    return self.exists(
      lambda state: isinstance(state,CommandDur.Succeed) and
        state.code == code and state.nr == nr)

  def transition(self, event):
    match event:
      case {Col.ID:Val.DISPATCH,Col.SCLK:t,Col.Code:c,Col.Nr:n}:
        return CommandDur.Succeed(t, c, n)
      case {Col.ID:Val.FAILURE|Val.SUCCESS,Col.Code:c,Col.Nr:n}
      if not monitored(c, n):
          return pc.error()

  @pc.data
  class Succeed(pc.HotState):
    sclk: str; code: str; nr: str

    def transition(self, event):
      match event:
        case {Col.ID:Val.CMD_SUCCESS|Val.CMD_FAILURE,
          Col.SCLK:t,Col.CmdCode:self.code,Col.CmdNr:self.nr}:
          self.statistics.record_duration(self.code,self.sclk,t)
          return pc.info(...)
```

Fig. 13. The command execution monitor.

format statistics such as time periods of autopsy windows, rates of change for each channel, and file compression rates.

Figure 14 shows fragments of this monitor. The main transition function (lines 2-10) creates different kinds of states, depending on what the incoming event is. Specifically an `EventBegun` (line 6, expiring on a 15 second timeout in line 19) when a channel is read, and a `DumpBegun` (line 10) when a autopsy window end has been detected (expiring on a 60 second timeout in line 28). Note that rate event monitoring is not initiated in the 60 seconds *after* the autopsy *recording off* event, ensured by a call of the `dumping` function (defined in a similar manner as the `monitoring` function in Figure 13 line 11) in the transition condition (line 5). Note how we record each 60 second autopsy window (line 9) upon detecting

the end of the window with a *recording off* event. Due to the fact that we only know the windows at their end, we need to re-access all information produced during monitoring once we know the windows. PyContract stores all messages produced with the methods `error` and `info` internally, which can be extracted with a call of the method `get_all_messages()`. These are then processed again, this time taking the now known windows into account. The fact that we have to process the messages again illustrates a weakness in the PyContract library wrt. expressing past time properties. Note, however, that even if PyContract could express past time properties conveniently, there is still the data analysis aspect which complicates matters.

```
class EHARates(pc.CSVMonitor):
  def transition(self, event):
    match event:
      case {Col.TYPE:Val.EHA,Col.ID:c,Col.SCLK:t,Col.DATA:d}
      if not self.dumping():
        return EHARates.EventBegun(c, float(d), float(t))
      case {Col.ID:Val.HEALTH_AUT_RECORDING_OFF, ...}
      if Options.AUTOPSY:
        windows.add_window(float(sclk))
        return EHARates.DumpBegun(float(sclk))

  @pc.data
  class EventBegun(pc.State):
    channel: str; value1: float; sclk1: float

    def transition(self, event):
      match event:
        case {Col.ID:self.channel,Col.SCLK:t,Col.DATA:d}
        if float(t) - self.sclk1 >= 15:
          ...; return pc.info(...)

  @pc.data
  class DumpBegun(pc.State):
    sclk: float

    def transition(self, event):
      match event:
        case {Col.SCLK:t} if float(t) - self.sclk > 60:
          ...; return pc.ok
```

Fig. 14. The sample rate monitor.

4 Conclusion

We presented an application of the RV library PyContract in Python to the analysis of log files from NASA's Europa Clipper flight computer. The analysis had as purpose to evaluate functional as well as non-functional (performance) properties. The effort demonstrates how such a temporal formalism can be used for data analysis, where the objective is not only to produce Boolean yes/no verdicts as in classical runtime verification, but also to produce richer forms of data. PyContract supports writing temporal properties. Adding data analysis to these becomes easy due to the fact that PyContract is a Python library, allowing to mix temporal specifications with code. Current work includes further development of the web-based interface using Dash, allowing easier construction and application of monitors as plugins. The interface allows to select monitors and logs to which they are applied. The logs are extracted from a database. The interface allows convenient browsing (filtering and coloring) of logs as well as visualization and tabulation of results. Wrt. longer term future work, there are rich opportunities for log analysis visualization.

References

1. D. Ancona, L. Franceschini, A. Ferrando, and V. Mascardi. RML: Theory and practice of a domain specific language for runtime verification. *Science of Computer Programming*, 205:102610, 05 2021.
2. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
3. H. Barringer and K. Havelund. TraceContract: A Scala DSL for trace analysis. In *Proc. 17th Int. Symposium on Formal Methods (FM 2011)*, volume 6664 of *LNCS*, pages 57–72, Limerick, Ireland, 2011. Springer.
4. H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. In *Proc. of the 7th Int. Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 111–125, Vancouver, Canada, 2007. Springer.
5. D. A. Basin, F. Klaedtke, S. Marinovic, and E. Zălinescu. Monitoring of temporal first-order properties with aggregations. *Formal Methods in System Design*, 46(3):262–285, 2015.
6. A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Proc. of the 7th Int. Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 126–138, Vancouver, Canada, 2007. Springer.
7. R. Castano, T. Vaquero, F. Rossi, V. Verma, E. V. Wyk, D. Allard, B. Huffmann, E. M. Murphy, N. Dhamani, R. A. Hewitt, S. Davidoff, R. Amini, A. Barrett, J. Castillo-Rogez, M. Choukroun, A. Dadaian, R. Francis, B. Gorr, M. Hofstadter, M. Ingham, C. Sorice, and I. Tierney. Operations for autonomous spacecraft. In *2022 IEEE Aerospace Conference (AERO)*. IEEE, Mar 2022.
8. K.-T. Cheng and A. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *30th ACM/IEEE Design Automation Conference*, pages 86–91, 1993.

9. C. Colombo, G. J. Pace, and G. Schneider. LARVA — safer monitoring of real-time Java programs (tool paper). In *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, SEFM '09, pages 33–37, Washington, DC, USA, 2009. IEEE Computer Society.
10. D. Dams, K. Havelund, and S. Kauffman. A Python library for trace analysis. In T. Dang and V. Stolz, editors, *22nd International Conference on Runtime Verification (RV)*, volume 13498 of *LNCS*, page 264–273. Springer International Publishing, 2022.
11. D. Dams, K. Havelund, and S. Kauffman. Runtime verification as documentation. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering (ISoLA)*, volume 13702 of *LNCS*, pages 157–173. Springer International Publishing, 2022.
12. B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. LOLA: Runtime monitoring of synchronous systems. In *Proceedings of TIME 2005: the 12th International Symposium on Temporal Representation and Reasoning*, pages 166–174. IEEE, 2005.
13. Dash. `https://plotly.com/dash`.
14. Daut. `https://github.com/havelund/daut`.
15. N. Decker, M. Leucker, and D. Thoma. Monitoring modulo theories. *Software Tools for Technology Transfer (STTT)*, 18(2):205–225, 2016.
16. Europa Clipper mission. `https://europa.nasa.gov`.
17. M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison-Wesley, 2010.
18. J. Gibbons and N. Wu. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 339–347, New York, NY, USA, 2014. Association for Computing Machinery.
19. S. Hallé and R. Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Transactions on Services Computing*, 5(2):192–206, 2012.
20. K. Havelund. Data automata in Scala. In *2014 Theoretical Aspects of Software Engineering Conference, TASE 2014, Changsha, China, September 1-3, 2014*, pages 1–9. IEEE Computer Society, 2014.
21. K. Havelund. Rule-based runtime verification revisited. *Software Tools for Technology Transfer (STTT)*, 17(2):143–170, 2015.
22. S. Kauffman, K. Havelund, and R. Joshi. nfer - a notation and system for inferring event stream abstractions. In *Runtime Verification - 6th Int. Conference, RV'16*, volume 10012 of *LNCS*, pages 235–250. Springer, 2016.
23. M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a run-time assurance tool for Java. In *Proc. of the 1st Int. Workshop on Runtime Verification (RV'01)*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.
24. E. Kurklu and K. Havelund. A flight rule checker for the LADEE Lunar spacecraft. In *17th International Colloquium on Theoretical Aspects of Computing (ICTAC'20)*, volume TBD of *LNCS*, 2020.
25. P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, pages 249–289, 2011.
26. Pandas. `https://pandas.pydata.org`.
27. PlantUML. `http://plantuml.com`.
28. PyContract. `https://github.com/pyrv/pycontract`.
29. Python pattern matching. `https://peps.python.org/pep-0636`.

30. G. Reger, H. C. Cruz, and D. Rydeheard. MarQ: Monitoring at runtime with QEA. In C. Baier and C. Tinelli, editors, *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015)*, pages 596–610. Springer, 2015.