Runtime Verification as Documentation

Dennis Dams[1], Klaus Havelund[2⋆], and Sean Kauffman[3]

[1] ESI (TNO), The Netherlands
[2] Jet Propulsion Laboratory, California Inst. of Technology, USA
[3] Aalborg University, Denmark

Abstract. In runtime verification, a monitor is used to return a Boolean verdict on the behavior of a system. We present several examples of the use of monitors to instead *document* system behavior. In doing so, we demonstrate how runtime verification can be combined with techniques from data science to provide novel forms of program analysis.

1 Introduction

The phrase "show me the code"[1] is well known amongst programmers, and reflects the need, and even desire, to read code in order to understand how it works. However, code can be arbitrarily complex and hard to comprehend, even for the most experienced. Program documentation, after all, remains the best way to convey information between humans about how a program works. Unfortunately, many software products are created without proper documentation. We can distinguish between at least three kinds of users of documentation: the programmer who wants to modify the code, the programmer who wants to use the code (e.g. a user of Python's `matplotlib` library), and finally the person that just wants to use the code as an application (e.g. a user of `www.github.com`).

When we think of documentation, we usually think of what we shall refer to as *static documentation*. Static documentation is ideally written once and does not change unless improved or modified due to changes in the software. Static documentation includes, but is not limited to, comments in the code. From comments in a special format, API documentation can be generated, as e.g. with JavaDoc [25], meant for programmers using the code as a library. Static documentation can also take the form of user guides or requirements documents. Documentation can be graphical as well as text-based. For example, sequence diagrams [38] are commonly used in requirements documents to illustrate the typical (happy-flow) behavior of an application. Graphical documentation can even be generated automatically from the code, as shown in [39]. Formalized specifications of the code can also function as documentation. This includes tests (unit, functional, or integration). That is, every pair of (input, expected output)

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

[1] The full quote, "Talk is cheap. Show me the code.", is attributed to Linus Torvalds.

provides a concrete, easy-to-understand example of how the code is supposed to behave. Formal specifications of code fragments, such as specifications in JML [8] can also function as documentation. The key point is that there are static artifacts that explain the code in a different manner than the code itself.

We shall, however, focus on what we shall refer to as *dynamic documentation*, by which we mean documentation of *program executions*. Producing debugging output can be considered dynamic documentation. However, one can put extra emphasis on producing comprehensible and helpful output in debugging mode as well as in production. In this paper we shall further narrow the scope and approach this topic with a starting point in the field of *Runtime Verification* (RV). Runtime verification is a broad field effectively covering any analysis of program/system executions. However, a common focus in literature is the verification of a program execution against a *formal specification* to determine whether the execution is well-formed, resulting in a Boolean (true,false) verdict. Examples of RV systems include [32,31,10,3,5,4,21,36,17,2]. We shall illustrate how one can go beyond the Boolean verdict domain by providing three examples of how runtime verification can be integrated with the broader field of data analysis, for the purpose of documenting program executions. Note that this augmentation of classical runtime verification from the Boolean domain to rich data domains has been studied in other work. In [7] the MFotl logic of the MonPoly tool was augmented with aggregators for computing sums, averages, etc over traces. Another approach is found in stream processing tools [14,19,12,37,18] which from streams of inputs (including the trace) produce new streams of data. Here, we illustrate different approaches using both standard data analysis tools and runtime verification tools provided as Python [34] libraries.

From a documentation point of view, a monitor specification provides a succinct piece of *static* documentation of the required behavior of a program. For example, in a tool like CommaSuite [11], program behavior is specified at the level of interfaces. Interface signatures show which methods are available, and possibly which callbacks are to be expected on an interface. As such, these document the "statics" of the interface. Interface protocols are state machines that specify the order in which calls and callbacks are allowed to occur. This is also static documentation. However, executing such a monitor provides dynamic documentation of the interface.

When running a monitor, either on a log previously produced by an executing program, or in parallel with an executing program, it will (in the basic verification case) complain when the execution does not satisfy the property specified. In this case, an explanation may be produced as well, e.g. the part the log which caused the failure. If no failures are encountered, the monitor will only report "all ok". Whether it fails or not on an execution is useful information. However, it may also be useful to get additional information during nominal executions.

In this paper, we present three examples of how runtime verification (understood in the classical verification sense) can be augmented to perform *data analysis*, which, according to [15] is the "*process of inspecting, cleansing, transforming, and modelling data with the goal of discovering useful information,*

informing conclusions, and supporting decision-making". Formulated differently, we shall produce data in rich data domains, including visualizations, rather than just Boolean verdicts.

The paper is organized as follows. We begin in Section 2 by discussing the dynamic documentation problem of state reconstruction, where a sequence of events are used to reconstruct the state of a program. We provide a simple example of state reconstruction using Python and then discuss examples we have observed in industrial practice. This leads us to a more complex example of dynamic documentation in Section 3, where we demonstrate visualization of events using the RV tool, PyContract [13]. The example has been examined using classical, Boolean valued, RV before, but we extend this analysis to examine the cause of failures and learn something new about how they occur. Finally, in Section 4, we show how the RV tool, nfer [28], can be used for debugging a timing problem in a program. We present an example where inconsistent timing would be difficult to debug using traditional methods and show how documenting the problem is possible using an RV tool and its visualization capabilities. We conclude in Section 5.

2 State Reconstruction with Python

A common use of dynamic documentation in industrial practice is for *state reconstruction*. A log file often records events rather than state, i.e. it shows what *happens* instead of what *is*. When analyzing a log file, typically in order to identify the root cause of an observed anomaly or failure, it is helpful to be able to inspect the state of a system at various points during execution. In such a case, a runtime monitor can be used to reconstruct the system state in between every two events recorded in a log file.

2.1 Robot tracking example

We use a simple example to illustrate this. Consider a robot that moves along the points of an (x, y) grid. It receives commands such as $(W, 6)$, telling it to move in western direction for 6 meters. These commands are recorded in a log file, which may then start as follows:

Time	Direction	Distance
00:00	N	5
01:00	E	2
02:00	S	8
03:00	W	4

In order to reconstruct the position of the robot after every command, a simple state machine is defined, shown (in Python syntax) in Figure 1. Note that this is an Extended Finite State Machine: its states are positions (x, y), with the initial state being $(0, 0)$. Its transition relation (`move`) takes an command consisting of a direction and a distance, and updates the state accordingly.

```
class Pos:
    x: int = 0
    y: int = 0

    def __repr__(self):
        return f"({self.x}, {self.y})"

    def move(self, vec):
        direction = vec[0]
        distance = vec[1]
        if direction == "N":
            self.y += distance
        elif direction == "E":
            self.x += distance
        elif direction == "S":
            self.y -= distance
        elif direction == "W":
            self.x -= distance
        return (self.x, self.y)
```

Fig. 1: State machine tracking a robot's position

We can feed this state machine with a sequence of commands and print the resulting position, as follows:

```
commands = [("N", 5), ("E", 2), ("S", 8), ("W", 4)]
pos = Pos()
for cmd in commands:
    pos.move(cmd)
print(pos)
```

which outputs

```
(-2, -3)
```

In order to decorate the log file shown above, the state machine is fed each of the events from the log, and the resulting state is added in an additional column in the log file:

Time	Direction	Distance	Position
00:00	N	5	(0, 5)
01:00	E	2	(2, 5)
02:00	S	8	(2, -3)
03:00	W	4	(-2, -3)

2.2 Other examples from industrial practice

In embedded systems, software interacts with, and often controls, the operation of physical (mechanical or electronic) components. When analyzing logs of sys-

tems, e.g. to track down the root cause of a malfunction, it is useful to be able to inspect the state of the physical components at any given point in the log file. In our interactions with software developers we have come across several examples of this. Here, we give some examples coming from a log analysis tool that is being used in the production of control software for a piece of high-tech equipment that consists of several components.

Each component has multiple led indicator lights. The leds can be in one of the modes off, on, or blinking. The commands that control the leds (`TurnOn(led`$_i$`)`, `TurnOff(led`$_i$`)`, `SetBlinking(led`$_i$`)`) are logged. At any point in the log file, the mode of each led can be known by finding the most recent command for this led. The log analysis tool used by the engineers uses a simple state machine, whose state indicates the modes of all leds, and whose transitions fire to update the state at every occurrence of one of the mentioned commands. The tool produces an interactive table showing the full log, in which clicking on a particular row (log line) opens a separate view displaying the sequence of all states that the collective leds assume. This view is again a table in which there is a column for each led, showing its mode after every change of state.

One component of the machine is a multi-segment robot arm that can move in 3 dimensions. The commands that control the overall position of the arm are logged, so that its position in space at any moment can be reconstructed by adding up the commands so far, much like in the introductory example in Section 2.1. In this case, the state of the arm is displayed by using a visualization in a 3D modeling environment. Any point in the log file can be selected, a button clicked, and the animated arm position shows up instantly in a pop-up window. This gives engineers a powerful tool to understand the behavior of the system, which is crucial in root cause analysis.

In our conversations with engineers, we realized that "the state of the system" has a different meaning to different people, usually reflecting their specific domain of expertise. For the mechanical engineer, the state of the system may be the position of the robot arm, while for the user interaction designer it may be the position of the joysticks and pedals that are used to operate the system. The abstract notion of a state machine to monitor "what the system is doing" at any point in time caters for all these different views of state.

3 Visualization with PyContract

We shall illustrate the transition from classical Boolean valued runtime verification to data analysis with a case study originally presented in [6]. The case study concerns a data-collection campaign performed by Nokia [1]. The campaign was launched in 2009, and collected information from cell phones of approximately 180 participants. The data collected were inserted into three databases DB1, DB2, and DB3, as shown on Figure 2. The phones periodically upload their data to database DB1. Every night, a script copies the data from DB1 to DB2. The script can execute for up to 6 hours. Furthermore, triggers running on DB2 anonymize and copy the data to DB3, where researchers can access and analyze

the anonymized data. These triggers execute immediately and take less than one minute to finish. The participants can access and delete their own data using a web interface to DB1.

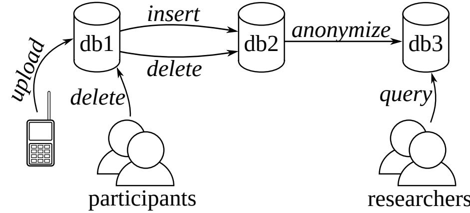


Fig. 2: Nokia's data-collection campaign

This is a distributed application producing events in different locations that then have to be merged into one log. The log produced, consisting of these and other events, contains 218 million events. In the case where two events have the same time stamp, there is no way to know which event comes before the other in the merged log. This is referred to as a *collapse of an interleaving* in [6], and leads to some intricate temporal properties.

The collected data must satisfy certain policies for how data are propagated between databases. One of these is shown in Figure 3, expressed in the first-order linear temporal logic Mfotl [7]. The property states that data (different from *unknown*) inserted in DB1 must be inserted into DB2 within 30 hours (by a script), unless it is being deleted from DB1 before then. Note that the deletion from DB1 or insertion in DB2 may occur at the same time (within the same second) as the insertion in DB1 but appear earlier due to the *collapse of an interleaving* problem. Hence we have to check for these events "within a second in the past" as well as 30 hours in the future.

$$\begin{aligned}
&\Box\, \forall user \cdot \forall data \cdot insert(user, db1, data) \wedge data \neq unknown \rightarrow \\
&\quad \blacklozenge_{[0,1s)} \Diamond_{[0,30h)}\ \exists user' \cdot \\
&\qquad insert(user', db2, data) \vee delete(user', db1, data)
\end{aligned}$$

Fig. 3: The Ins_{1_1} property from [7]

We first present a classical Boolean verdict monitor of this property, and then subsequently augment this monitor with data analysis. These monitors are written using the RV tool, PyContract [13], a library for monitoring in Python with an automata-like syntax. In [22] the case study is analyzed with the runtime verification tool Daut [16,20] and the static analysis tool Cobra [24,9]. Our PyContract monitors resemble the Daut monitor in [22].

3.1 The Boolean verdict monitor

We assume the definition of two class constructors `Ins`(*time*, *user*, *db*, *data*) and `Del`(*time*, *user*, *db*, *data*) for representing insertion and deletion events such as `Ins(1000,"Adda",Db.ONE,3742)` and `Del(2000,"Adda",Db.ONE,3742)`. Figure 4 shows a Boolean verdict monitor in PyContract for the Ins_{1_1} property. First the PyContract library is imported (line 1), and its contents subsequently referred to prefixed with `pc`. A monitor, in this case `Verifier`, in PyContract is defined as a class extending the class `pc.Monitor`. The monitor defines a main `transition` function (lines 4-12), and two explicitly named states: `I2D1` (lines 14-22) and `Track` (lines 24-35), both of which are parameterized with a `time` stamp and a `data` value.

The main `transition` function (lines 4-12) is always enabled. It takes an event as its argument and returns a new state[2]. Pattern matching[3] is conveniently used to determine which event is being submitted. The first case states that if we observe an insertion into DB2 or a deletion from database DB1, then we create a `I2D1` state, parameterized with the time `t` and the data `d`, representing the fact that that data was inserted in DB2 or deleted from DB1. This is our way of remembering the past. Note that when matching against a dotted name, such as `Db.ONE` and `Db.TWO`, the incoming value has to match exactly. In contrast, an unqualified name, such as `t` and `d` (i.e. a bare name with no dots), is interpreted as a capture pattern, binding the incoming value to the name. The second case covers the situation where the observed event is an insertion into DB1. In this case, if there exists a `I2D1(t, d)` fact in the fact memory (line 9), with the same time stamp `t` as the insertion (due to the *collapse of an interleaving* problem) and the same data `d`, then the property is satisfied for this data, and we return `pc.ok`, causing the state to be removed from the monitor memory. Note how a state object, here `Verifier.I2D1(t, d)` can be used as a Boolean condition. Otherwise we return a `Track(t,d)` state (line 12), which will track the data `d`.

The `I2D1` state, (lines 14-22) extends the `State` class. The state also defines a `transition` function which returns `pc.ok` upon observing an event that is more than a second apart from the `time` stamp that was passed as argument, thereby causing the state to be removed from memory. This is done for efficiency reasons to avoid storing unnecessary states. Note that a state does not need to contain a transition function, in which case it becomes a persistent state, just a data record representing a fact about the past.

The `Track` state (lines 24-35) extends `HotState` meaning that it is an error for such a state to exist in the monitor memory at the end of the trace, and represents the property that some event needs to occur. The state also contains a `transition` function. The first case observes an event with a time stamp more than 30 hours (108,000 seconds) since the insertion of the `data` in database DB1. This represents the violation of the property. The second case covers the correct insertion of the data into DB2 or deletion from DB1 within the 30 hours.

[2] PyContract generally permits returning a list of states.

[3] Pattern matching was introduced in Python version 3.10 [35].

```
import pycontract as pc

class Verifier(pc.Monitor):
    def transition(self, event):
        match event:
            case Ins(t, _, Db.TWO, d) | Del(t, _, Db.ONE, d):
                return Verifier.I2D1(t, d)
            case Ins(t, _, Db.ONE, d) if d != '[unknown]':
                if Verifier.I2D1(t, d):
                    return pc.ok
                else:
                    return Verifier.Track(t, d)

    @pc.data
    class I2D1(pc.State):
        time : int
        data : str

        def transition(self, event):
            match event:
                case e if e.time - self.time > 1:
                    return pc.ok

    @pc.data
    class Track(pc.HotState):
        time: int
        data: str

        def transition(self, event):
            match event:
                case e if e.time - self.time > 108000:
                    return pc.error('30 hours have passed')
                case Ins(_, _, Db.TWO, self.data) |
                     Del(_, _, Db.ONE, self.data):
                    return pc.ok
```

Fig. 4: The Boolean verdict monitor code

Running the monitor on the first 2 million events in the log causes several error messages to be issued, triggered by the 30 hour deadline being passed. The following is an example of such an error message:

```
*** error transition in Verifier:
    state Track(1276508300, '96563613')
    event 255 Ins(time=1277200698, user='script',
                  database=<Db.TWO: 2>, data='66935671')
    time since insertion: 692398 seconds
```

The error message is due to an insertion into DB1 followed by an event more than 30 hours (108,000 seconds) later, without an insertion of the same data in DB2 in between. A `Track` state was created at time `1276508300` due to the insertion of data `'96563613'` into DB1. The error message is a result of subsequently encountering an `Ins` (insert) event (number 255) at time `1277200698` into database DB2 of some other data. The latter event indicates that more than 30 hours have passed since the insertion into DB1 of the data `'96563613'`.

3.2 Augmenting the Boolean verdict monitor for visualization

We may now be interested in better understanding the pattern of these violations. This can be achieved by e.g. visualizing the time, referred to as duration, from when a piece of data is inserted into DB1 until it is either inserted into DB2 or deleted from DB1. We augment our monitor with data analysis, going beyond the Boolean domain. Note that we can easily do this in PyContract since a monitor is just a Python class and we can use all the features that Python supports, including numerous data analysis libraries.

Figure 5 shows the augmented monitor. We first define an `__init__` function (lines 2-4), which initializes a monitor local variable `durations`, which is a list of tuples $(time, dur)$ where dur is an observed duration from when a piece of data is inserted into DB1 until it is *resolved*: inserted into DB2 or deleted from DB1, and $time$ is the time of the resolve. We then augment the `transition` function in the `Track` state. Specifically we add a statement (lines 18-19) recording the duration from insertion into DB1 to failure caused by 30 hours having passed, and a statement 23-24), recording the duration from insertion into DB1 to insertion into DB2 or deletion from DB1.

At the end of monitoring the first 2 million events of the log we can now process the `durations` variable. The method `to_graph`[4] (line 6) processes this variable and graphs the durations (y-axis) as a function of time (x-axis). It also introduces a horizontal line representing the average duration and a line representing the 30 hour deadline. The result is shown in Figure 6a. It shows that all the durations are above the 30 hour deadline and that there really only are two time points where these errors are reported (the thick dots). This is not the pattern we expected of the first two million events in the log, where we would have expected that some durations are within the 30 hour deadline. To understand this better we also plotted the number of `Track` states, as can be seen in Figure 6b. This shows that there is an initial creation of `Track` states, then a release of all these causing the first set of errors, and again immediately a new boost of creations, and then again a release, causing errors. The visualization has thus given us some better understanding of the work of the monitor, beyond Boolean verdicts.

To illustrate the expected result, we created an artificial log of 10,000 events, with insertions into DB1 occurring randomly time wise, and deletions from DB1

[4] We have not shown the contents of this function. It is 10 lines of code, and uses Python's `matplotlib.pyplot`, `pandas`, and `statistics` libraries.

and insertions into DB2 occurring randomly according to a Gaussian distribution reflecting the frequency of the script that updates DB2. The resulting durations are shown in Figure 7a and the number of active states are shown in Figure 7b.

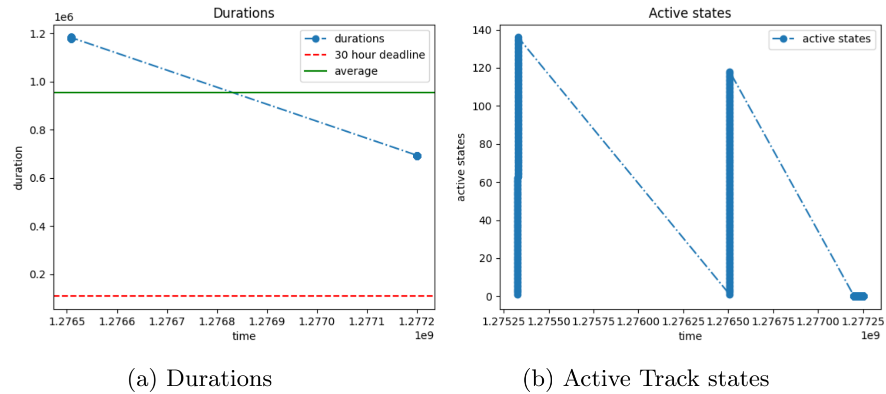
```
class Verifier(pc.Monitor):
    def __init__(self):
        super().__init__()
        self.durations: List[tuple[int,float]] = []

    def to_graph(self): ...

    ...

    @pc.data
    class Track(pc.HotState):
        time: int
        data: str

        def transition(self, event):
            match event:
                case e if e.time - self.time > 108000:
                    self.monitor.durations.append(
                        (e.time, e.time - self.time))
                    return pc.error(f'30 hours have passed')
                case Ins(_, _, Db.TWO, self.data) |
                     Del(_, _, Db.ONE, self.data):
                    self.monitor.durations.append(
                        (e.time, e.time - self.time))
                    return pc.ok
```

Fig. 5: The monitor code augmented for failure analysis

4 Timing Debugging with nfer

In this section we show how an RV tool built for analyzing event traces may be applied to annotate and debug the execution of a program. We choose as an example a popular open-source Python program called xhtml2pdf that takes a URI as an input and converts the website at that address to a local PDF file [23]. This tool is a good example because it performs a complex task and, because it must download the contents of a website, involves substantial input/output. The input/output is useful for our purposes because different parts of the program must wait for downloads to finish, leading to interesting timing behavior. We used xhtml2pdf version 0.2.8, cloned from the project git repository.

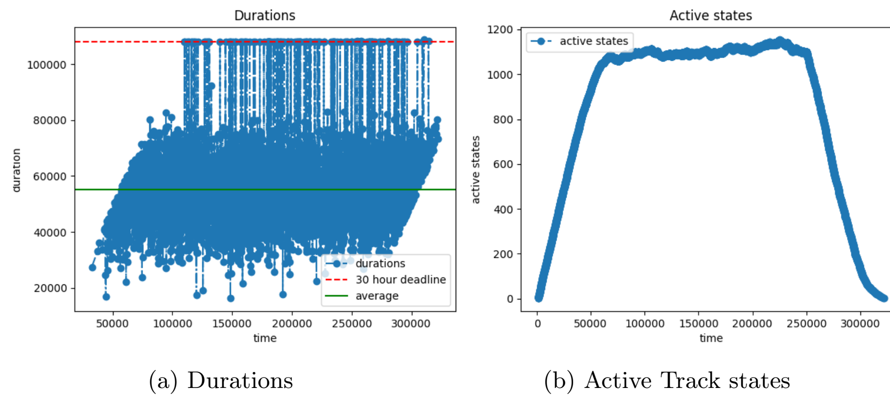
To perform the analysis, we used the Python interface of the RV tool nfer. Nfer is a language and tool for the abstraction and analysis of event traces [28,29].



(a) Durations

(b) Active Track states

Fig. 6: Graphs for first 2 million events of Nokia log.



(a) Durations

(b) Active Track states

Fig. 7: Graphs for test log with 10,000 events.

It was originally designed to improve human and machine comprehension of telemetry from the Mars Science Laboratory (MSL) [33], more commonly known as the Curiosity Rover. Although the language was originally intended for offline (batch) analysis, its implementation supports online monitoring and includes an integration with the Python language [27]. The nfer Python module can be installed using PyPi, Python's package manager [26].

We now consider an example where we must analyze the timing behavior an execution of the xhtml2pdf program to debug a problem. In this example, we have introduced a timing delay in the program when a certain piece of data is encountered. Specifically, in the `NetworkFileURI` class, we added a delay to the `extract_data` method when it encounters a photo of Klaus Havelund, one of the authors of this paper. While the problem in this case is contrived, such errors occur in real programs and pose challenges for developers. This type of problem

is difficult to debug for a number of reasons: 1. the problem is intermittent and depends on data, so it is hard to reproduce reliably, 2. using a debugger to find the problem does not work because it will interrupt the timing of the application, and 3. if the timing of the function is variable for other reasons then adding logging may not provide enough information to isolate the execution in question.

We now show how nfer can be used to debug such an intermittent timing problem. First, the program must be instrumented so that its execution can be recorded. Functions may be instrumented using the `watch` decorator, or, to instrument an entire package or module, nfer provides the `instrument` function. Once the program is instrumented, its execution can be visualized using the web-based Graphical User Interface (GUI) distributed with nfer.

The code that instruments and visualizes xhtml2pdf is shown in Figure 8 starting on line 7. The code is inserted prior to calling the main entry point of the program, a function called `execute`. Lines 7 and 8 import the `instrument` and `gui` functions from nfer while line 10 instruments the classes and functions in the `xhtml2pdf` package and line 11 launches the GUI in a web browser. The `gui` function is non-blocking and causes a web server to run in a separate process, allowing it to mostly avoid interfering with the timing of the program.

```
1  def command():
2      if "--profile" in sys.argv:
3          print ("*** PROFILING ENABLED")
4          # profiling code removed from figure
5      else:
6          ## nfer instrumentation and visualization ##
7          from nfer.instrument import instrument
8          from nfer.gui import gui
9
10         instrument("xhtml2pdf")
11         gui()
12         ############################################
13         execute()
```

Fig. 8: Instrumenting and visualizing a program execution with nfer

The nfer `instrument` function works using Python's `inspect` module to analyze the environment at runtime. It iterates over loaded packages and modules, looking for callable functions and methods and instrumenting them using the nfer decorator, `watch`. The `watch` decorator wraps a function so that, when it is called, a timer captures the timestamps before and after it executes. The decorated function execution is then reported to nfer as an interval, with the arguments and return value of the function associated as data elements.

As these intervals are reported, they are visualized in the nfer GUI with an update delay of at most one second. The result of running such an instru-

mented copy of xhtml2pdf on the website `http://havelund.com` can be seen in Figure 9. In the figure, five interval names corresponding to five functions have been selected for display. The main display shows each execution of the functions on a timeline, where overlapping executions indicate that they happened concurrently. For example, the bottom interval in the figure (in blue) shows an execution of the `pisaParser` function that began shortly after second *:36* and ended shortly before second *:44*. The GUI has been zoomed out to show the whole execution using the timeline at the bottom of the figure, where the green box shows the visible portion of the timeline.

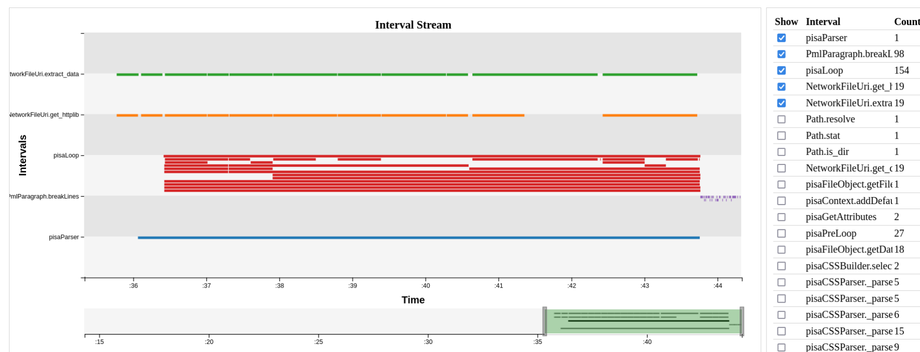


Fig. 9: Nfer GUI from the instrumented xhtml2pdf

The figure contains a clue as to the execution of `extract_data` with the timing delay. Each of the top (in green) intervals correspond to executions of `extract_data`, while all of the intervals immediately below (in orange) correspond to executions of the `get_httplib` function. In the figure, each `extract_data` interval is matched by a `get_httplib` interval except in one case where the `get_httplib` interval appears to end early. This clearly visualizes the timing problem. By examining the interval in the GUI (not shown), we can see that the shorter `get_httplib` execution was passed an argument of `http://havelund.com/havelund.jpg`. It is important to note that this is not possible to conclude by examining the delayed `extract_data` call on its own, since it is not passed the URI as an argument.

In many cases, the automatically instrumented function calls may not be sufficient to find a timing delay such as the one shown above. For example, if the function with the delay was executed many millions of times, finding one instance with a simple visual inspection could be difficult. Nfer allows the user to write problem-specific rules that define custom intervals. These rules define intervals using relationships between other, existing intervals including the automatically generated ones. Figure 10 shows how to add an nfer rule using the Python interface as an alternative to the automatic instrumentation visualized above. In the figure, line 2 tells nfer to look for executions of `NetworkFileUri.get_httplib`

and give them the abbreviation `get` for easier reference (done with the `get:` notation). Line 3 then adds that we are only interested in executions of `get_httplib` that occur during executions of `NetworkFileUri.extract_data`. These executions are abbreviated as `extract`. On line 4, the matched executions are filtered to include only those where the `extract_data` execution continues more than 100 milliseconds after the `get_httplib` execution ends. Line 5 tells nfer that the generated intervals should include a piece of data named "resource" that contains the arguments to `get_httplib`, and line 6 names the created interval `delayed` so it is easier to identify in the GUI. The nfer formalism is designed to be concise and expressive [30] and its Python interface supports its complete syntax.

```
1        monitor(
2          when("get:NetworkFileUri.get_httplib")
3          .during("extract:NetworkFileUri.extract_data")
4          .where("extract.end - get.end > 100")
5          .map("resource", "get.args")
6          .name("delayed")
7        )
```

Fig. 10: Monitoring an nfer rule

Figure 11 shows the nfer GUI, zoomed in on a shorter time span and only showing the `get_httplib`, `extract_data`, and `delayed` intevals. In the figure, the same suspicious gap is visible in duration between the execution of the `extract_data` (on the top line, in orange) and `get_httplib` (on the second line, in blue) functions. Now, however, the third line (in purple) shows an instance of the `delayed` interval that nfer reported due to the rule in Figure 10. The image shows how hovering the mouse over the `delayed` interval displays the resource that caused the delay.

5 Conclusion

In this work we presented the notion of dynamic documentation in the form of data analysis anchored in runtime verification. Some runtime verification tools can process and output more information than pure Boolean verdicts, and these capabilities may be leveraged for documenting program executions. We showed three examples of this idea, demonstrating state reconstruction, failure analysis, and timing debugging, using trace visualization as a tool. These examples demonstrate that dynamic documentation using runtime verification is both possible and effective. More work can be done to develop the concept of dynamic documentation. For one, theories should be developed and tested on what kinds of dynamic documentation are helpful when developing and maintaining programs. Other technologies are relevant, for example using machine learning techniques such as specification mining, which can be thought of as a form of data analysis.

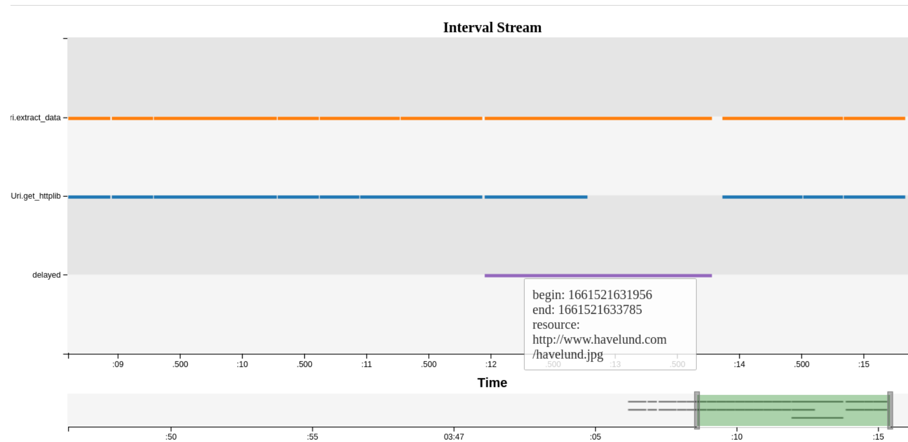


Fig. 11: Nfer GUI zoomed in and showing the additional annotation

References

1. I. Aad and V. Niemi. NRC data collection campaign and the privacy by design principles. In *Proceedings of the International Workshop on Sensing for App Phones (PhoneSense'10)*, 2010.
2. 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.
3. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
4. 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.
5. 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.
6. D. Basin, M. Harvan, F. Klaedtke, and E. Zalinescu. Monitoring usage-control policies in distributed systems. In *Proc. of the 18th Int. Symp. on Temporal Representation and Reasoning*, pages 88–95, 2011.
7. 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.
8. P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with jml and esc/java2. In *Formal Methods for Components and Objects*, pages 342–363. Springer, 2006.
9. Cobra on github. `https://github.com/nimble-code/Cobra`, 2020.
10. 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.

11. CommaSuite. https://projects.eclipse.org/projects/technology.comma.
12. L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, and D. Thoma. TeSSLa: Temporal stream-based specification language. In *Formal Methods: Foundations and Applications*, volume 11254 of *LNCS*, pages 144–162. Springer, 2018.
13. D. Dams, K. Havelund, and S. Kauffman. A Python library for trace analysis. In *Runtime Verification - 22nd Int. Conference, RV'22, Proceedings*, LNCS. Springer, 2022.
14. 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.
15. Data analysis, wikipedia. https://en.wikipedia.org/wiki/Data_analysis.
16. Daut. https://github.com/havelund/daut.
17. N. Decker, M. Leucker, and D. Thoma. Monitoring modulo theories. *Software Tools for Technology Transfer (STTT)*, 18(2):205–225, 2016.
18. P. Faymonville, B. Finkbeiner, M. Schwenger, and H. Torfah. Real-time stream-based monitoring, 2019.
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. K. Havelund and G. Holzmann. Programming event monitors. May 2022. Submitted to journal, under review.
23. D. Holtwick. xhtml2pdf PyPi website. https://pypi.org/project/xhtml2pdf/, 2022.
24. G. J. Holzmann. Cobra: a light-weight tool for static and dynamic program analysis. *Innov. Syst. Softw. Eng.*, 13(1):35–49, 2017.
25. Javadoc documentation. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html.
26. S. Kauffman. PyPi NferModule. https://pypi.org/project/NferModule/.
27. S. Kauffman. nfer – a tool for event stream abstraction. In *International Conference on Software Engineering and Formal Methods (SEFM'21)*, volume 13085 of *LNCS*, pages 103–109. Springer, 2021.
28. 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.
29. S. Kauffman, K. Havelund, R. Joshi, and S. Fischmeister. Inferring event stream abstractions. *Formal Methods in System Design*, 53:54–82, 2018.
30. S. Kauffman and M. Zimmermann. The complexity of evaluating nfer. In *International Symposium on Theoretical Aspects of Software Engineering (TASE'22)*, volume 13299 of *LNCS*, pages 388–405. Springer, 07 2022.
31. K. G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using Uppaal. In *Formal Approaches to Software Testing*, volume 3395 of *LNCS*, pages 79–94. Springer, 2004.
32. 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. http://dx.doi.org/10.1007/s10009-011-0198-6.

33. MSL - Mars Science Laboratory. `https://science.jpl.nasa.gov/projects/msl`.
34. Python. `http://www.python.org`.
35. Python pattern matching. `https://peps.python.org/pep-0636`.
36. 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.
37. C. Sánchez. Online and offline stream runtime verification of synchronous systems. In *Runtime Verification*, pages 138–163, Cham, 2018. Springer.
38. UML sequence diagram tutorial. `https://www.lucidchart.com/pages/uml-sequence-diagram`.
39. R. von Hanxleden, E. A. Lee, H. Fuhrmann, A. Schulz-Rosengarten, S. Domrös, M. Lohstroh, S. Bateni, and C. Menard. Pragmatics twelve years later: A report on Lingua Franca. In *Proc. of the 11th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2022)*, LNCS. Springer, 2022. in this volume.