International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-023-00706-1

GENERAL

Regular

Programming event monitors

Klaus Havelund[1] · Gerard J. Holzmann[2]

Accepted: 30 May 2023

Abstract

Specification languages for runtime verification are commonly rooted in formal languages, such as temporal logic, automata, or regular expressions. We argue that, for practical purposes, specification languages for monitoring should allow language features similar to those found in general purpose programming languages, in addition to providing specialized monitoring constructs. Using a realistic and large event-log, we compare two such programming-oriented monitoring language systems to a temporal logic-based monitoring system that was previously evaluated on the same log. The first programming-oriented language is a library in Scala developed for runtime verification. The other language is a scripting language, originally developed for fast static code analysis. We formulate the same reasonably complex properties as in the temporal logic case, using both methods, and compare the efficiency with which they can be checked against the large event log, and the ease with which the properties can be formulated.

Keywords Runtime verification · Log analysis · Specification language · Temporal logic · Domain-specific language · Scala · C

1 Introduction

Runtime Verification (RV) facilitates monitoring the execution of a system against a formal specification of a property, commonly to detect violations. The system emits events to a monitor, which then updates its internal state and emits a message or informs the system in case the property is violated. Typically in state-of-the-art RV systems, events are records carrying data. Numerous RV systems have been developed in the past, including, e.g., [3, 4, 7, 10, 11, 13, 16, 18, 21, 22, 24, 26, 28], to mention only a few. Most of these systems support writing properties in formal languages such as temporal logic, state machines, regular expressions, grammars, rule systems, or stream processing, to mention the most common. These formal languages offer very succinct notations. An attempt to support several common temporal patterns in one language, focusing on ease of writing and reading specifications, is described in [8]. Most of the languages are defined as external DSLs (Domain-Specific Languages), also referred to as “little languages” with their own grammar and parser, or deep internal DSLs, where the user builds an AST (Abstract Syntax Tree) of the specification using a general purpose programming language. In both cases, the expressive power of the specification language is exactly the expressive power of the “little” DSL.

It is, however, our experience that in practice there is often a need to *program* monitors, using more powerful language constructs known from general purpose programming languages. For example, this occurs when complex data processing is needed based on the data observed in events. Furthermore, in some cases, it may be desirable for a monitor to produce a *richer data product* than just a Boolean valued verdict, including even trace visualization and general data analysis. Our thesis consequently is that writing monitors requires a specification language that is Turing complete, allowing for arbitrary programming when needed, but with syntax that also allows reasonably succinct specifications in cases where the problem is “simple” enough. In other words, we believe that there is a need for monitor specification languages in the space between formal languages at the one end and general purpose programming languages at the other end. We refer to such languages as *programming logics*. There exist other attempts in this direction, including the stream-based HStriver [15], a deep Haskell DSL allowing Haskell types to be used in the DSL.

✉ K. Havelund
klaus.havelund@jpl.nasa.gov

G.J. Holzmann
gholzmann@acm.org

1 Jet Propulsion Laboratory, California Institute of Technology, Pasadena, USA

2 Nimble Research, Monrovia, USA

Published online: 22 June 2023

The authors have in previous work developed two such programming logics, Daut and Cobra. Daut [17] is an internal shallow DSL in Scala, effectively a Scala library for runtime verification. Daut was developed to provide the user with the expressive power and succinctness of Scala while at the same time supporting writing a combination of temporal properties and state machines. Cobra [19] was developed as a static analysis tool, offering an external scripting-like DSL, with its own grammar for writing source code queries. It was developed with execution speed in mind to allow writing such queries over large code bases and get them executed within seconds. Cobra was later extended for dynamic analysis (runtime verification) for the work presented in this paper.

We present a case study comparing Daut and Cobra to one particular instance of these formal language-based frameworks, namely MonPoly [6], and its temporal logic MFOTL (Metric First-Order Temporal Logic) that supports past and future time operators as well as data aggregation operators. MFOTL properties are translated to automata-based monitors. The case study concerns properties of a large data-set that was published by Nokia. In the paper [6], the same data-set was used, with properties specified in the MFOTL logic and analyzed with the MonPoly tool. We apply Daut and Cobra to the same data-set and compare with the results presented in [6], both wrt. performance and wrt. succinctness of specifications. Note that we do not compare Daut and Cobra to MFOTL wrt. expressiveness. The general result is that Daut and Cobra both outperform MonPoly, likely due to the lower-level programming approach (and Cobra outperforms Daut). On the other hand, the MonPoly specification is, not surprisingly, more succinct.

The paper is organized as follows. Section 2 describes the case-study and the data-set used, as well as the investigated properties stated in MonPoly's MFOTL temporal logic. Section 3 describes the application of Daut to the case study, and Sect. 4 describes the application of Cobra. Section 5 presents the performance measurements of applying the monitors to the Nokia log, and discusses the specifications and the efforts required to construct them. Finally, Sect. 6 concludes the paper.

2 The Nokia log and its expected properties

Our case study was presented in [6] and concerns a realistic 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 was inserted into three databases DB1, DB2, and DB3, as shown in Fig. 1 (from [6]). 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. Participants can access and delete their own data using a web interface to DB1. This is a distributed application producing events in different locations that then have to be merged into one log.

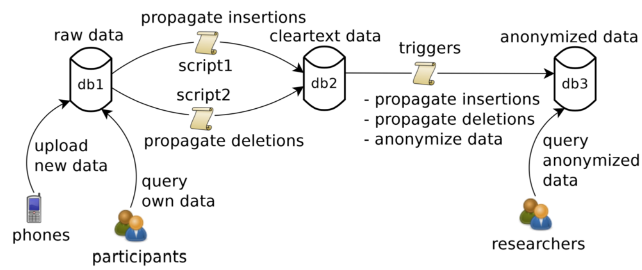

Fig. 1 Nokia's Data-collection Campaign (from [6])

The log produced, consisting of these and other events, contains 218 million events, which is substantial size. This log is the result of merging logs from different log producers, as explained in [6]. The merging respects time stamps, all in seconds, in the sense that in the merged log, one event e_1 from one source will come before another event e_2 from another source if e_1's time stamp is strictly less than e_2's time stamp. However, in the case where the two events have the same time stamp, there is no way to know which event comes before the other in the merged log. The order could be $e_1 e_2$ as well as $e_2 e_1$. It is only guaranteed that events with the same time stamps are grouped together in the merged log. This is referred to as a *collapse of an interleaving* in [6], and leads to some intricate temporal properties, as we shall see below, and is part of the challenge addressed in [6].

Collected data must satisfy certain policies, including policies for how data are propagated between databases. A total of 14 policies are presented in [6]. We have focused on two of these, as shown in Fig. 2, expressed in the first-order linear temporal logic MFOTL [7]. The properties concern the following two database operations (data are identified by a unique ID):

- *insert*(*user*, *db*, *data*): insertion of *data* into *db* by *user*.
- *delete*(*user*, *db*, *data*): deletion of *data* from *db* by *user*.

Property Ins_1_2 states that data inserted in DB1 must be inserted into DB2 within 30 hours (by a script, which is a kind of user), unless it is deleted from DB1 before then. The time limit is 30 hours since the script runs every 24 hours and takes up to 6 hours to execute.

Property Del_1_2 is more complicated and states that if data is deleted from DB1, one of two things should be true: either it is deleted from DB2 within 30 hours, or we have the situation where the data was inserted into DB1 within the past 30 hours, has not been inserted in DB2 since then, and will not be inserted within the next 30 hours. In that second case, there is no need to delete the data from DB2.

Ins_1_2:
$\Box\, \forall user \cdot \forall data \cdot insert(user, db1, data) \wedge data \neq unknown \rightarrow$
$\blacklozenge_{[0,1s)} \Diamond_{[0,30h)}\ \exists user' \cdot$
$insert(user', db2, data) \vee delete(user', db1, data)$

Del_1_2:
$\Box \forall user \cdot \forall data \cdot delete(user, db1, data) \wedge data \neq unknown \rightarrow$
$(\blacklozenge_{[0,1s)} \Diamond_{[0,30h)}\ \exists user' \cdot delete(user', db2, data)) \vee$
$((\Diamond_{[0,1s)} \blacklozenge_{[0,30h)}\ \exists user' \cdot insert(user', db1, data)) \wedge$
$(\blacksquare_{[0,30h)} \Box_{[0,30h)}\ \neg \exists user' \cdot insert(user', db2, data)))$

Fig. 2 The properties Ins_1_2 and Del_1_2

Let us now explain the formulas in Fig. 2 in more detail. MFOTL is a first-order linear time temporal logic with future and past time operators annotated with time intervals. Let $\mathbb{N}$ denote the set of natural numbers. An interval has the form $[a,b)$ where $a \in \mathbb{N}$, $b \in \mathbb{N} \cup \{\infty\}$, and $a < b$. It denotes the set $\{x \in \mathbb{N} \mid a \leq x \wedge x < b\}$. MFOTL formulas can be composed of predicates, such as $insert(user, db, data)$ and $delete(user, db, data)$, equality and inequality between terms, Boolean operators, universal and existential quantification over data, and the temporal operators (amongst others): $\Box_{[a,b)}\varphi$ (always φ in the future within the interval $[a,b)$), $\Diamond_{[a,b)}\varphi$ (sometime φ in the future within the interval $[a,b)$), $\blacksquare_{[a,b)}\varphi$ (always φ in the past within the interval $[a,b)$), and $\blacklozenge_{[a,b)}\varphi$ (sometime φ in the past within the interval $[a,b)$). We also have that $\Box\varphi = \Box_{[0,\infty)}\varphi$. Note that in the case of the past time operators, the interval $[a,b)$ means that a is closest to "now" and b represents a point in time further back.

Ins_1_2 states that if there is an $insert(user, db1, data)$ event where $data \neq unknown$, then within less than a second in the past or within 30 hours in the future, a user inserts the data in DB2 or it is deleted from DB1. Note that "within less than a second in the past" effectively means "now", since the smallest observable time unit is 1 second. The consequent of Ins_1_2 is:

$$\blacklozenge_{[0,1s)} \Diamond_{[0,30h)}\ \exists user' \cdot insert(user', db2, data) \vee delete(user', db1, data)$$

Formula $\blacklozenge_{[0,1s)} \Diamond_{[0,30h)} \psi$ is equivalent to $\blacklozenge_{[0,1s)} \psi \vee \Diamond_{[0,30h)} \psi$. The subformula $\blacklozenge_{[0,1s)} \psi$ takes care of the situation where ψ (insertion in DB2 or deletion from DB1) might occur with the same time stamp as the insertion in DB1 (the antecedent of Ins_1_2), but in the merged log occurs right before, as discussed above. The property is in [6] referred to as *collapse-sufficient* in the sense that it does not yield false positives and false negatives when monitoring the collapse of an interleaving, where events with the same time stamps are merged to be next to each other, but in unknown order.

The property Del_1_2 is collapse-sufficient as well. The property Del_1_2 should now (with some effort) be comprehensible observing that:

$$\Diamond_{[0,1s)} \blacklozenge_{[0,30h)} \psi = \Diamond_{[0,1s)}\ \psi \vee \blacklozenge_{[0,30h)}\ \psi$$

$$\blacksquare_{[0,30h)} \Box_{[0,30h)}\ \psi = \blacksquare_{[0,30h)}\ \psi \wedge \Box_{[0,30h)}\ \psi$$

3 Log analysis with Daut

Daut (an acronym for: 'Data automata') [12, 17] is a library in the Scala [25, 29] programming language for monitoring event sequences. Programming a monitor in Daut consists effectively of writing a Scala program using the library. This results in a framework that combines the expressiveness of a modern programming language with the domain-specific features. Scala combines object-oriented and functional programming and supports the definition of *internal DSLs*. An internal DSL is a library, but defined such that the use of the library feels like programming in a domain-specific language. The library supports a notation that combines state machines with temporal logic. States and formulas can be parameterized with data, allowing monitoring of events that carry data, as we shall see.

Daut is derived from TraceContract [2, 31], also a Scala library for monitoring, which was used for verifying command sequences sent to NASA's LADEE (Lunar Atmosphere And Dust Environment Explorer) spacecraft during its entire mission [5, 23]. Daut has furthermore been integrated in the Mesa tool [30], which supports concurrent monitors communicating with message passing, using Scala's actor model, to check for properties specified using Daut. As a case study, Mesa has been used to monitor flights from live US airspace data streams.

In a bird's eye view of Daut, a monitor is created and utilized as described below. We first determine what type of events, say `Event`, the monitor shall monitor. This can be any type. Then, the monitor is defined as a subclass, extending the `Monitor[E]` class, that is parameterized with the event type `E`. We then create an instance of the user defined monitor class and feed it with events, one by one. Finally, we terminate the monitor. Events can be fed directly as shown here, read in from a file, e.g., in a loop, or even produced by a running program.

```
type Event = ...

class MyMonitor extends Monitor[Event] {
  ... // specification to be monitored
}

object Main {
  def main(args: Array[String]): Unit = {
    val m = new MyMonitor
    m.verify(event1)
    m.verify(event2)
     ...
    m.end()
  }
}
```

```
trait Ev {val t: Long}
case class Ins(t:Long,u:String,db:Db,d:String) extends Ev
case class Del(t:Long,u:String,db:Db,d:String) extends Ev

trait Db
case object Db1 extends Db
case object Db2 extends Db
```

Fig. 3 The Daut event type

The monitor will issue error messages when violations are detected. The user can also define a call-back function, which is called in case of violations, performing any desirable actions.

3.1 Events

We first have to define our type of events. The CSV log contains two kinds of events, namely insertions and deletions, each occupying one line of comma-separated terms, as shown below, with two line breaks inserted for each event to fit the printed format.

```
insert, tp = 349, ts = 1277200700,
        u = script,  db = db2,
        p = 1, d = 66932652
delete, tp = 392593, ts = 1279883112,
        u = user18, db = db1,
        p = 172, d = 79131680
```

We need to represent the time stamps (`ts`), the user (`u`), the database (`db`) being inserted into or deleted from, and the data (`d`) being inserted or deleted. We define our event type to be the trait[1] `Ev`, see Fig. 3, which contains a time value `t` (we use short names for presentation purposes to be able to fit the article format, the actual implementation contains longer, more explanatory names). Then we define two subclasses of `Ev`, namely `Ins` and `Del`, representing insertions and deletions, along with their constructor arguments: time stamp, user, database and data. The trait `Db` and its two objects represent the two databases we are concerned about. When the CSV file is parsed, objects of type `Ins` and `Del` are generated and fed to our monitors.

3.2 Property Ins_1_2

The monitor corresponding to property Ins_1_2 is shown in Fig. 4. Before we dive into its contents, we explain briefly the computation model of Daut. The internal memory of a monitor is a set of states[2] sub-classing the `state` trait (not shown). Such a state contains a user-defined transition function. When an event is submitted to the monitor, each state in the memory is *applied* to the event, resulting in zero, one, or more new states to be generated. There are different kinds of states inspired by temporal logic operators [27]. These states differ in (1) how they react to an event that does not match any transition (staying in the state, failing, or droping the state), (2) how they react to an event that matches a transition to another state (remaining in the source state or leaving it), and (3) how they are evaluated at the end of the trace (true or false). We can now explain how the property is modeled.

```
1  class Ins_1_2 extends Monitor[Ev] {
2    val hrs_30 = 108000
3    val sec_1 = 1
4
5    case class I2D1(t: Long, d: String) extends state {
6      watch {
7        case e if e.t - t > sec_1 => ok
8      }
9    }
10
11   always {
12     case Del(t, _, Db1, d) => I2D1(t, d)
13     case Ins(t, _, Db2, d) => I2D1(t, d)
14
15     case Ins(t, _, Db1, d) if d != "[unknown]" =>
16       if (exists { case I2D1(`t`, `d`) => true }) ok
17       else
18         hot {
19           case e if e.t - t > hrs_30 => error
20           case Ins(_, _, Db2, `d`) => ok
21           case Del(_, _, Db1, `d`) => ok
22         }
23   }
24 }
```

Fig. 4 Daut monitor for property Ins_1_2

3.2.1 Past time

The property requires us to keep track of past insertions into db2 and deletions from db1. The property, however, does not require us to record which of the two happened. Therefore, we define a state `I2D1` (lines 5-9) to represent any of these events. It is parameterized by the time `t` of insertion or deletion and the data `d` inserted or deleted. The transition function of the state is modeled by the call of the function `watch` in lines 6-8: it takes as argument a partial function of type: `PartialFunction[Ev,Set[state]]`. Such partial functions can in Scala be defined as a list of case statements. The function is defined for any event matching any (in this case one) of the "transitions". In this case, the state will trigger on any event `e` where the time value is more than 1 second away from the parameter value `t` of the state. That is, once such a state has been generated, it automatically "goes away again" after a next event is observed with a bigger time stamp.

The `watch` function in addition defines a kind of state as discussed above: (1) it stays in the monitor's memory in case no transition triggers, (2) it leaves the state in case a transition is triggered, and (3) it is a final state, meaning that it does not cause an error at the end of monitoring when the

[1] A trait is like a class by encapsulating method and field definitions, but it can contain undefined methods. Unlike class inheritance, in which each class must inherit from just one superclass, a class can extend any number of traits.

[2] In the basic case it is a set of states. However, states can be organized in a key-indexed map, supporting more efficient state lookup.

Table 1 The different kinds of states supported by Daut. For each kind of state it is indicated how it behaves if its transition function does not match an incoming event, how it behaves if there is a match, and whether it is an error if such a state exists at the end of monitoring

State	If no match	If match	At end
always	stay	stay	ok
watch	stay	leave	ok
hot	stay	leave	error
wnext	error	leave	ok
next	error	leave	error

end() function is called. The specification contains three such transition-defining functions: watch, always, and hot. In addition, Daut offers the transition-defining functions (not used in this presentation): next and wnext (weak next). The behaviors of the transition functions are shown in Table 1.

The monitor itself is controlled by an always state, lines 11-23, which continuously watches new events (if a transition triggers, the always state stays in the memory). The first two cases in the transition function, lines 12 and 13, trigger on deletions from Db1 and insertions into Db2. In both cases, I2D1 state is generated and added to the monitor memory. The last transition is described in the next section.

3.2.2 Future time

The last transition of the always function, lines 15-22, triggers in line 15 when an insertion into Db1 is observed, where the data is not unknown. To better understand lines 16-22, observe that the following equivalence holds for the Ins_1_2 subformula that occurs after the implication $\rightarrow$ in Fig. 2:

$$
\begin{aligned}
&\blacklozenge_{[0,1s)} \Diamond_{[0,30h)}\ \exists user' \cdot\\
&\quad insert(user', db2, data) \vee delete(user', db1, data)\\
&=\\
&\blacklozenge_{[0,1s)}\ \exists user' \cdot\\
&\quad insert(user', db2, data) \vee delete(user', db1, data)\\
&\vee\\
&\Diamond_{[0,30h)}\ \exists user' \cdot\\
&\quad insert(user', db2, data) \vee delete(user', db1, data)
\end{aligned}
$$

That is, we have split the formula into a past time and a future time formula. We check the past in line 16 by checking whether there exists any I2D1 state in the monitor memory with the values t and d which were matched in line 15. The quotes around these names in line 16 express that they should match the previously defined values and not be binding occurrences. If such a match exists, the ok state is returned, which terminates the monitoring corresponding to this particular Db1 insertion. Otherwise (**else**), we check the future by entering a hot state (meaning that the monitor needs to exit this state before the end of monitoring), which we can leave in one of three ways: (1) if an event occurs with a time stamp past 30 hours, or if before that an insertion into Db2 or deletion from Db occurs of the data.

3.3 Property Del_1_2

The programming of the Daut monitor for property Del_1_2 is not as direct as in the case of the Ins_1_2 property. We need to perform a sequence of rewrites of the original formula to reach a formula suitable for coding in Daut. We will go through these rewrites in the following. For the purpose of the presentation we assume a function $[\![\_]\!] : LTL \rightarrow Daut$ from LTL to Daut, as it applies to this particular example.[3] The resulting Daut monitor is shown in Fig. 5.

Consider the original Del_1_2 property in Fig. 2. First, we define some short-hands for non-temporal predicates occurring in the formula, with names suggestive of the operations they represent and the database they operate on:

$$
\begin{aligned}
d_1 &= delete(user, db1, data) \wedge data \neq unknown\\
d_2 &= \exists user' \cdot delete(user', db2, data)\\
i_1 &= \exists user' \cdot insert(user', db1, data)\\
i_2 &= \exists user' \cdot insert(user', db2, data)
\end{aligned}
$$

We can now write the Del_1_2 property as follows:

$$\Box \forall user \cdot \forall data \cdot d_1 \rightarrow \varphi$$

where φ is the formula:

$$
\begin{aligned}
&(\blacklozenge_{[0,1s)} \Diamond_{[0,30h)}\ d_2) \vee\\
&((\Diamond_{[0,1s)} \blacklozenge_{[0,30h)}\ i_1) \wedge (\blacksquare_{[0,30h)} \Box_{[0,30h)}\ \neg i_2))
\end{aligned}
$$

Line 21 in Fig. 5 corresponds to the antecedent of the implication (the left part of $\rightarrow$). Wrt. the right-hand side φ, we observe the following equivalences:

$$
\begin{aligned}
\blacklozenge_{[0,1s)} \Diamond_{[0,30h)}\ d_2 &= \blacklozenge_{[0,1s)}\ d_2 \vee \Diamond_{[0,30h)}\ d_2\\
\Diamond_{[0,1s)} \blacklozenge_{[0,30h)}\ i_1 &= \Diamond_{[0,1s)}\ i_1 \vee \blacklozenge_{[0,30h)}\ i_1\\
\blacksquare_{[0,30h)} \Box_{[0,30h)}\ \neg i_2 &= \blacksquare_{[0,30h)}\ \neg i_2 \wedge \Box_{[0,30h)}\ \neg i_2
\end{aligned}
$$

With these equivalences we can rewrite the right-hand side formula φ as follows (naming the second disjunct ϕ):

$$
\begin{aligned}
&(\blacklozenge_{[0,1s)}\ d_2 \vee \Diamond_{[0,30h)}\ d_2) \vee\\
&\underbrace{((\Diamond_{[0,1s)}\ i_1 \vee \blacklozenge_{[0,30h)}\ i_1) \wedge (\blacksquare_{[0,30h)}\ \neg i_2 \wedge \Box_{[0,30h)}\ \neg i_2))}_{\phi}
\end{aligned}
$$

[3] We do not claim the existence of a general *elegant* translation function from LTL with future and past time operators to Daut. However, since Daut, as an extension of Scala, is Turing complete, a translation does exist.

```
1  class Del_1_2 extends Monitor[Ev] {
2    val hrs_30 = 108000
3    val sec_1 = 1
4
5    case class D(t:Long, db:Db, d:String) extends state {
6      watch {
7        case event if event.t − t > sec_1 ⇒ ok
8      }
9    }
10
11   case class I(t:Long, db:Db, d:String) extends state {
12     watch {
13       case event if event.t − t > hrs_30 ⇒ ok
14     }
15   }
16
17   always {
18     case Del(t, _, Db2, d) ⇒ D(t, Db2, d)
19     case Ins(t, _, db, d) ⇒ I(t, db, d)
20
21     case Del(t, _, Db1, d) if d != "[unknown]" ⇒
22       if (exists { case D(`t`, Db2, `d`) ⇒ true }) ok
23       else {
24         val s1 =
25           if (exists { case I(t0, Db1, `d`) ⇒
26             t − t0 ≤ hrs_30 }) ok
27           else hot {
28             case e if e.t − t > hrs_30 ⇒ error
29             case Del(t1, _, Db2, `d`)
30               if t1 − t ≤ hrs_30 ⇒ ok
31             case Ins(`t`, _, Db1, `d`) ⇒ ok
32           }
33         val s2 =
34           if (exists { case I(t0, Db2, `d`) ⇒
35             t − t0 ≤ hrs_30 })
36             hot {
37               case e if e.t − t > hrs_30 ⇒ error
38               case Del(t1, _, Db2, `d`)
39                 if t1 − t ≤ hrs_30 ⇒ ok
40             }
41           else hot {
42             case e if e.t − t > hrs_30 ⇒ ok
43             case Del(t1, _, Db2, `d`)
44               if t1 − t ≤ hrs_30 ⇒ ok
45             case Ins(t1, _, Db2, `d`)
46               if t1 − t ≤ hrs_30 ⇒
47                 hot {
48                   case e if e.t − t > hrs_30 ⇒ error
49                   case Del(t1, _, Db2, `d`)
50                     if t1 − t ≤ hrs_30 ⇒ ok
51                 }
52           }
53         (s1, s2)
54       }
55   }
56 }
```

Fig. 5 Daut monitor for property Del_1_2

3.3.1 Past time

We observe that three past time subformulas occur, namely $\blacklozenge_{[0,1s)}\, d_2$ and $\blacklozenge_{[0,30h)}\, i_1$ and $\blacksquare_{[0,30h)}\, \neg i_2$. We thus need to record and remember all insertions into Db1 and Db2 for 30 hours, as well as all deletions from Db2 for one second. This is managed by the declaration of the states D and I in lines 5-15, as well as their creation in lines 18-19. Furthermore, we observe that the leftmost formula $\blacklozenge_{[0,1s)}\, d_2$ is a past-time formula, which we can translate into an if-statement:

```
if (⧫[0,1s) d2) ok else ⟦◇[0,30h) d2 ∨ φ⟧
```

The resulting remaining formula $\Diamond_{[0,30h)}\, d_2 \vee \phi$ to be translated can be rewritten as follows using the distributive law of Boolean algebra $(p \vee (q \wedge r)) = (p \vee q) \wedge (p \vee r)$:

$$
\begin{aligned}
&\Diamond_{[0,30h)}\, d_2 \vee \phi = \\
&\quad \underbrace{(\Diamond_{[0,30h)}\, d_2 \vee \Diamond_{[0,1s)}\, i_1 \vee \blacklozenge_{[0,30h)}\, i_1)}_{\psi_1} \\
&\quad \wedge \\
&\quad \underbrace{(\Diamond_{[0,30h)}\, d_2 \vee (\blacksquare_{[0,30h)}\, \neg i_2 \wedge \Box_{[0,30h)}\, \neg i_2))}_{\psi_2}
\end{aligned}
$$

The subformulas ψ_1 and ψ_2 are represented in Fig. 5 by the states s1, lines 24-32, and s2, lines 33-52, respectively. The result returned is the set of those two states, represented by the tuple in line 53. Both states have to lead to success (cannot fail), corresponding to a conjunction.

Let us dive into the definition of s1 and s2. Both states are defined using an if-statement with a condition corresponding to the past time formulas respectively $\blacklozenge_{[0,30h)}\, i_1$ and $\blacksquare_{[0,30h)}\, \neg i_2$, occurring in ψ_1 and ψ_2, respectively (the latter must be negated in the if-statement's condition):

```
val s1 = if (⧫[0,30h) i1) ok else ⟦◇[0,30h) d2 ∨ ◇[0,1s) i1⟧
val s2 = if (¬■[0,30h) ¬i2) ◇[0,30h) d2 else
         ⟦◇[0,30h) d2 ∨ □[0,30h) ¬i2⟧
(s1,s2)
```

The formulas to translate for the then-parts of the two if-statements should be obvious by examining the if-conditions and ψ_1 and ψ_2. We now proceed with the else-parts.

3.3.2 Future time

For the else-parts, in the case of s1, we need to check the formula $\Diamond_{[0,30h)}\, d_2 \vee \Diamond_{[0,1s)}\, i_1$, which results in the hot state in lines 27-32. Here we wait for either a deletion from Db2 within 30 hours or an insertion into Db1 in less than 1 second, effectively meaning *this* second, namely the same time as t. In case an event occurs, line 28, with a time passing 30 hours, a violation has been detected.

The else-part of s2, lines 41-52, modeling the formula $\Diamond_{[0,30h)}\, d_2 \vee \Box_{[0,30h)}\, \neg i_2$, is slightly more complicated and can be read as follows. If at any time 30 hours have passed without any insertions into Db2, all is well. If an insertion into Db2, however, occurs, we continue to monitor for any deletions from Db2 to occur, in which case all is well.

3.3.3 An adjustment

It turns out that the Del_1_2 monitor in Fig. 5 is very inefficient due to the large amounts of I states (insertions) that need to be stored and remembered for 30 hours. We therefore

```
class History(resetBound: Int, timeLimit: Long) {
  val map = collection.mutable.Map[String, Long]()
  var counter : Int = 0

  def get(d: String): Option[Long] = map.get(d)

  def put(d: String, t : Long) : Unit = {
    counter += 1
    if (counter == resetBound) {
      counter = 0
      map.filterInPlace {
        case (_,t0) ⇒ t − t0 ≤ timeLimit
      }
    }
    map.put(d, t)
  }

  def within(d: String, now: Long): Boolean = {
    get(d) match {
      case None ⇒ false
      case Some(t) ⇒ now − t ≤ timeLimit
    }
  }
}
```

Fig. 6 Data structure for recording the past

had to re-program the recording of the past differently. Figure 6 shows the class History for recording updates (insertions or deletions) to one of the databases as a hashmap from data to time stamps representing the time they were inserted or deleted. The history will get cleaned up at every resetBound update to the hashtable, removing all entries older than a given timeLimit, e.g., 30 hours. The method within(d, now) returns true if data d was entered in the history within the timeLimit from the current time now.

Figure 7 shows the modified monitor for the Del_1_2 property using the History class. The changes compared to the monitor in Fig. 5 are the lines:

- 6-8 (replacing lines 5-15 in Fig. 5), declaring objects of the History class.
- 11-13 (replacing lines 18-19 in Fig. 5), updating the History objects.
- 16, 19, and 27 (replacing lines 22, 25-26, and 34-35 in Fig. 5), calling the within method.

3.4 Monitor execution

Once we have defined our monitors, we can combine them into one parent monitor, named Monitors, as shown in Fig. 8. Monitors can generally be combined hierarchically in this manner, as a way of grouping monitors. Figure 9 shows the main program creating an instance monitor of Monitors, reading from the CSV file via an instance of the LogReader class, and feeding the generated events to the monitor. As we shall see, the LogReader class filters out irrelevant events, hence there can be more events (csvFile.hasNext is true), but no more relevant events remain in the log file, resulting in csvFile.next to return None. Note also that we pass the line number to the monitor for error reporting purposes.

Daut detects 82,886 violations of the Ins_1_2 property and 25 violations of the Del_1_2 property. Each error is reported by indicating which event caused the monitor to track an event, the triggering event, and which event actually caused the monitor to report a violation. As an example,

```
1  class Del_1_2_opt extends Monitor[Ev] {
2    val hrs_30 = 108000
3    val sec_1 = 1
4    val sec_0 = 0
5
6    val I1 = new History(500000, hrs_30)
7    val I2 = new History(500000, hrs_30)
8    val D2 = new History(500000, sec_0)
9
10   always {
11     case Ins(t, _, Db1, d) ⇒ I1.put(d, t)
12     case Ins(t, _, Db2, d) ⇒ I2.put(d, t)
13     case Del(t, _, Db2, d) ⇒ D2.put(d,t)
14
15     case Del(t, _, Db1, d) if d != "[unknown]" ⇒
16       if (D2.within(d, t)) ok
17       else {
18         val s1 =
19           if (I1.within(d, t) ) ok
20           else hot {
21             case e if e.t − t > hrs_30 ⇒ error
22             case Del(t1, _, Db2, 'd')
23               if t1 − t ≤ hrs_30 ⇒ ok
24             case Ins('t', _, Db1, 'd') ⇒ ok
25           }
26         val s2 =
27           if (I2.within(d, t))
28             hot {
29               case e if e.t − t > hrs_30 ⇒ error
30               case Del(t1, _, Db2, 'd')
31                 if t1 − t ≤ hrs_30 ⇒ ok
32             }
33           else hot {
34             case e if e.t − t > hrs_30 ⇒ ok
35             case Del(t1, _, Db2, 'd')
36               if t1 − t ≤ hrs_30 ⇒ ok
37             case Ins(t1, _, Db2, 'd')
38               if t1 − t ≤ hrs_30 ⇒
39               hot {
40                 case e if e.t − t > hrs_30 ⇒ error
41                 case Del(t1, _, Db2, 'd')
42                   if t1 − t ≤ hrs_30 ⇒ ok
43               }
44           }
45         (s1, s2)
46       }
47   }
48 }
```

Fig. 7 Daut monitor for property Del_1_2, optimized

```
class Monitors extends Monitor[Ev] {
  monitor(new Ins_1_2, new Del_1_2_opt)
}
```

Fig. 8 Daut combining monitors into one

```
object VerifyLog {
  def main(args: Array[String]): Unit = {
    val csvFile = new LogReader("path/to/ldcc.csv")
    val monitor = new Monitors
    while (csvFile.hasNext) {
      csvFile.next match {
        case None ⇒
        case Some(e) ⇒ monitor.verify(e, csvFile.lineNr)
      }
    }
    monitor.end()
  }
}
```

Fig. 9 Daut main program

the following message reports violation number 144 of the Ins_1_2 property. The triggering event (matching the pattern in line 15 of Fig. 4) is event number 324, an insertion of data `96554472` at time `1276507789`, and the violating event is another insertion at time `1277200698`, which is nearly a week later. Hence no timely insertion into `Db2` or deletion from `Db1` of the data `96554472` was observed.

```
*** ERROR
trigger event: Ins(1276507789,[unknown],Db1,96554472)
  event number 324
current event: Ins(1277200698,script,Db2,66935671)
  event number 698
Ins_1_2 error # 144
```

3.5 Parsing CSV files

In this section, we shall briefly discuss how the CSV files were parsed and how the `Ev` events (see Fig. 3) were generated that were fed to the monitors. There are numerous libraries for parsing CSV files, and we chose FastCsv [14], claimed to be fast. Figure 10 shows a class `FastCSVReader` using this library, and providing effectively two methods `hasNext: Boolean`, returning true if there are more rows in the CSV file, and `next(): List[String]`, delivering the next row as a list of its columns. Figure 11 shows the `LogReader` class, which provides the same two methods, but where the method `next(): Option[Ev]` selects only events that are of interest, namely insertions and deletions, and only those concerning db1 and db2. It returns an optional event `e` of type `Ev` as `Some(e)` in case such a row is found, and `None` otherwise. Recall that a single row has the form:

`cmd, k1=v1,...,kn=vn`

The function `getData` takes as argument a single row, represented as the list List("cmd","k1=v1", ..., "kn=vn"), of its column elements, and returns a map from the keys to the values: Map("k1" → "v1", ..., "kn" → "vn").

```
class FastCSVReader(fileName: String) {
  import java.io.File
  import de.siegmar.fastcsv.reader.CsvReader
  import de.siegmar.fastcsv.reader.CsvRow
  import java.nio.charset.StandardCharsets
  import scala.collection.JavaConverters._

  val file = new File(fileName)
  val csvReader = new CsvReader
  val csvParser =
    csvReader.parse(file, StandardCharsets.UTF_8)
  var row: CsvRow = csvParser.nextRow()

  def hasNext: Boolean = row != null

  def next(): List[String] = {
    val line = row.getFields.asScala.toList
    row = csvParser.nextRow()
    line
  }
}
```

Fig. 10 Daut CSV parsing

```
class LogReader(fileName: String) {
  val reader = new FastCSVReader(fileName)
  val INSERT = "insert"
  val DELETE = "delete"
  var lineNr: Long = 0

  def getData(line: List[String]): Map[String,String] = {
    var map : Map[String,String] = Map()
    for (element ← line.tail) {
      val src_rng = element.split("=").map(_.trim())
      map += (src_rng(0) → src_rng(1))
    }
    map
  }

  def hasNext: Boolean = reader.hasNext

  def next: Option[Ev] = {
    var e: Option[Ev] = None
    breakable {
      while (reader.hasNext) {
        val line = reader.next()
        lineNr += 1
        val name = line(0)
        if (name == INSERT || name == DELETE) {
          val map = getData(line)
          val db = map("db")
          if (db == "db1" || db == "db2")  {
            val t = map("ts").toLong
            val u = map("u")
            val db = if (map("db") == "db1") Db1 else Db2
            val d = map("d")
            name match {
              case INSERT ⇒ e = Some(Ins(t, u, db, d))
              case DELETE ⇒ e = Some(Del(t, u, db, d))
            }
            break
          }
        }
      }
    }
    e
  }
}
```

Fig. 11 Daut Log reader

3.6 Indexing

Daut offers a capability for optimizing monitors using a simple indexing technique. The basic idea consists of structuring a monitor's memory as a hash table from *keys* to sets of states, where the user defines what the keys should be. The `Monitor` class contains the definition of the following function:

```
def keyOf(e: Ev): Option[String] = None
```

which for a given event returns the default `None` value, signifying that no key has been defined. The user can override this function, as shown in Fig. 12 for our example. We have in this function defined the keys to be the data part of the events. In the hash table, each datum (the key) is mapped to the set of states concerning that specific datum. When an event is submitted to the monitor, we just look up the set of concerned states and only process those. This can speed up monitoring considerably, as, for e.g., demonstrated in [30].

The idea of letting the programmer explicitly program the `keyOf` function, originally suggested for the Rust programming language by Rajeev Joshi [20], is related to the automated slicing approach supported by RV systems such as JavaMop

```
class IndexedMonitor extends Monitor[Ev] {
  override def keyOf(e: Ev): Option[String] = {
    e match {
      case Ins(_, _, _, d) ⇒ Some(d)
      case Del(_, _, _, d) ⇒ Some(d)
    }
  }
}

class Ins_1_2 extends IndexedMonitor { ... }
class Del_1_2 extends IndexedMonitor { ... }
```

Fig. 12 Daut indexing

[24] and MarQ [28]. In these systems, traces are sliced into subtraces, based on event arguments, where each subtrace is fed to a propositional monitor. The automated approach found in these systems is difficult to transfer to Daut due to the fact that it is an internal *shallow* DSL, where specifications are not easily analyzable without dealing with the Scala compiler.

However, we cannot use this indexing approach for the monitors shown. The reason is that these monitors refer to time, and their progress depends on events continuously being provided with time stamps in order for the monitors to "know what time it is". For example, consider line 19 in Fig. 4: `case e if e.t - t > hrs_30 ⇒ error`. This case will not trigger unless an event arrives with a time stamp beyond 30 hours. Although such an event may occur, it may not be for the same data, and hence it will be submitted to a different bucket in the hash table. Consequently, no error may be issued.

4 Log analysis with Cobra

The Cobra tool [9, 19] was designed as an interactive static analyzer for large source code archives, and is therefore an unusual choice for an application of a dynamic instead of static verification technique. The Cobra tool was designed to use highly efficient data-structures that can be queried quickly in interactive analyses of code bases. The tool was extended more recently with new options for processing also live data-streams, where the data structure being queried is maintained as a sliding window into a potentially unbounded event-stream read from standard input.

Cobra queries can be expressed in three main ways. A first method is to use the tool's interactive query language that lets us navigate the input stream and match on patterns of interest. This method is typically used in interactive sessions once a code archive has been read in core as a simple token-stream, with some minimal pre-processing. A second method is to use a powerful inline scripting language, interpreted in real-time, to resolve queries. A third and final method, which is the most efficient, is to write the queries in unrestricted C code as a back-end module that can be compiled and linked to the Cobra front-end, thus providing access to its data structures.

In this paper, we choose to use the interpreted scripting language to express the two properties from the Nokia study. This would appear to put the verification process at a disadvantage compared to tools that use compiled code for the queries. We will show, though, that this is not the case. We begin by exploring how the two fairly complex properties can be expressed in Cobra's scripting language.

4.1 Property Ins_1_2

The insert property, formalized in MFOTL in [6], requires that data inserted into database DB1 be migrated to database DB2 within 30 hours, unless the same data are deleted from DB1 first. To check this property, we must remember all insertion events for database DB1 for maximally 30 hours (108,000 seconds) and unless one of the two other events is found within that interval, flag a violation.

The Cobra checker script for this property is shown in Fig. 13. The script is executed once for every *token* in the input stream, with a token being created by the Cobra tool for each lexical symbol in the input stream. We rely on a small preprocessing step for the checkers that converts the somewhat redundant CVS format from the original log into the four relevant values we need to check the properties. The preprocessing step also handles the *time collapse* issue mentioned in [6] where the ordering of events with identical timestamps is undetermined. If an insert action into DB1 follows a delete action from DB1 or an insert action into DB2 for the same data id, we know that the insert action into DB1 must have come first.

The script uses a short-hand to match on tokens containing identifiers named *insert* or *delete*, in an if-then-else statement. If matched, the script then locates the timestamp information, the target database, and the unique data identifier that follow in the input stream. (Note that a # symbol followed by a space or tab is considered a comment.)

The dot refers to the current token being processed, and the two relevant fields of that token we are using here are named *.nxt*, which refers to the next token in the sequence, and *.txt*, which refers to the text field.

Once the four parameters have been located, we check the type of event. If it is an *insert*, we check the target database value. If it is DB1, we remember the insert event in an associative array that is indexed with the data identifier field and stores the timestamp plus 30 hours. This is the time limit for the migration of the data to DB2, unless it is deleted from DB1 first.

An insert event into DB2 causes the obligation to be deleted with an *unset* operation on the relevant element of the associative array, and similarly, if we see a *delete* event for DB1, we remove the obligation for that data item in the same way. The unset operation has no effect if the item is not present in the array.

```
%{
    if (#insert || #delete)
    {   event = .txt;
        . = .nxt; timestamp = .txt;     # timestamp
        . = .nxt;    db = .txt;         # db1/db2/db3
        . = .nxt; data = .txt;          # data id

        if (event == "insert")
        {       if (db == "db1")
                {       Obligation[data] = timestamp
                                               + 108000;
                } else
                {       if (db == "db2")
                        {       unset Obligation[data];
                }       }
        } else  # delete
        {       if (db == "db1")
                {       unset Obligation[data];
        }       }

        if (ots != timestamp)
        {       ots = timestamp;
                for (x in Obligation)
                {       if (Obligation[x.txt] > 0
                        &&  ots > Obligation[x.txt])
                        {   fails++;
                            print x.txt "␣fails\n";
                            Remove[x.txt] = 1;
                }       }
                for (x in Remove)
                {       unset Obligation[x.txt];
                }
                unset Remove;
        }
    }
%}
%{
        for (x in Obligation)
        {       fails++;
                print x.txt "␣unmet␣obligation\n";
        }
        print "number␣of␣violations:␣" fails "\n";
        Stop;
%}
```

Fig. 13 Cobra checker script for property Ins_1_2

Note also that variables or arrays need not be declared before they are used in a script, and their type is derived from the context in which they are used. Typically, that type will be a string, but it can also be an integer value or a token reference.

One final bit of processing is to remove obligations from the associative array that have passed the maximum time window waiting for the data migration or deletion. To do so, we check if the timestamp value has changed since the last time the script executed, and if so, we iterate through the elements of array *Obligation* to find the violations. Those elements can now be removed from the array. To avoid modifying the array while we are also iterating over the elements we remember the indices that can be omitted in a separate array called *Remove* and then delete those elements from array *Obligation* after the iteration is complete. That helper array can itself then also be removed once it has fulfilled its purpose.

The final part of the script in Fig. 13 runs after all tokens in the input stream have been processed (if that point ever comes). It checks if there are any remaining unmet obligations at the point of termination. If so, those will be reported. This final part of the script ends with a *Stop* command to indicate that we don't intend this part of the processing to be repeated for more tokens. The script execution terminates after the warnings and final tally of all violations have been reported.

4.2 Property Del_1_2

The script for checking the *delete* property requires some more processing, but the basic outline is very similar to that of the *insert* property.

The property requires that for every delete event of data from database DB1, either the same data is also deleted from database DB2 within a time window of 30 hours, or the data were inserted into DB1 within 30 hours earlier and has not yet migrated to DB2 since then, nor within the next 30 hours. This is a somewhat convoluted statement, when stated in English, but the check is fairly straightforward to encode in a Cobra script.

The main part of the script is shown in Fig. 14. We see the same initial match on the identifier names *insert* and *delete*, and the location of the three additional parameters we need. We can skip further processing if the target database is DB3, since operations on that database do not affect the property we are trying to check.

We must now maintain a sliding window of the last 30 hours (108,000 seconds) worth of events, which is done here with a call to function *add_window()* that we discuss shortly. Two other functions take care of the processing of the *delete* and *insert* events, and performing the related checks. The second script at the end is again used to check for any unmet obligations when the event stream ends, if indeed it does.

Function definitions and data initialization are performed in a startup script that runs first. The definition of the function

```
%{
        if (#delete || #insert)
        {   event = .txt;
            . = .nxt; timestamp = .txt; # timestamp
            . = .nxt;    db = .txt;     # db1/db2/db3
            . = .nxt; data = .txt;      # data id

            if (db == "db3")
            {   Next;
            }
            add_window(event, timestamp, db, data);
            if (event == "delete")
            {   handle_delete(timestamp, db, data);
            } else  # insert
            {   if (db == "db2")
                {   handle_insert(timestamp, data);
            }   }
        }
%}
%{
        cnt = 0;
        unset Obligation[0];
        for (x in Obligation)
        { cnt++;
          print cnt "␣data␣" x.txt "␣unmet␣obligation\n";
        }
        Stop;
%}
```

Fig. 14 Cobra checker script for property Del_1_2, main part

```
%{
    ...
    function add_window(tp, ts, dtb, dt)
    {       counter++;
            if (counter > 500000)   # slide window
            {       counter = 0;
                    slide_window(ts);
            }
            if (tp == "insert")     # only inserts
            {       n = list_tok(); # new list element
                    n.mark = ts;    # timestamp
                    n.seq  = dt;    # data id
                    if (dtb == "db1")
                    {       n.curly = 1; # db1
                    } else
                    {       n.curly = 2; # db2
                    }
                    list_append(window, n);
            }
    }
    # make sure these are global
    counter = 0;    # used by add_window
    q = .;          # used by handle_delete()
    n = .;          # used by add_window()
    x = .;          # used by migrated()
    Obligation[0]   = 0;
    Nonmigration[0] = 0;
    print "Starting\n";
    Stop;
%}
```

Fig. 15 Cobra checker script for property Del_1_2, initialization and function definition

add_window and the global data initialization is illustrated in Fig. 15.

The window that holds at least the last 30 hours worth of events is moved forward once every 500,000 calls to the *add_window()* function to reduce the overhead a bit. It does so by calling another function called *slide_window()*.

The script uses Cobra library functions for maintaining a list of tokens for the sliding window. We obtain a new token for this purpose with the call to *list_tok()* and we add it to a list named *window* with the call to *list_append()*. The list library is more expansive, but these calls suffice for what we need to do here.

Because we are working with the predefined token structures, rather than a user-defined data structure, we must store all the information we have into the available fields of that structure. Here we store the timestamp value in the field *n.mark*, the data id in field *n.seq* and we convert the database string into an integer that is stored in field *n.curly*.

Next, we define the two short functions *slide_window()* and *handle_insert()*, as shown in Fig. 16.

The definitions are fairly straight-forward. In function *slide_window()* we traverse the list of earlier relevant events and check if any are older than 30 hours. If so, we omit them from the list. The library function *list_top()* returns the element at the head of the list, but does not remove it, and the function *list_pop()* is used to remove the first element of the list.

Function *handle_insert()* checks if there is a prohibition on the migration of the data item into database DB2 (note in Fig. 14 that the function is only called for inserts that target this database). If a violation is found, it is reported

```
    function handle_insert(ts, dt)
    {   if (Nomigration[dt] != 0)
        {       # within 30 hr
                if (ts ≤ Nomigration[dt])
                {   print dt ":␣migrates␣deleted␣data\n";
                }
                unset Nomigration[dt];
        }
    }
    function slide_window(nts)
    {       n = list_top(window);
            while (n.seq > 0 && nts − n.mark > 108000)
            {       list_pop(window);
                    n = list_top(window);
            }
    }
```

Fig. 16 Cobra checker script for property Del_1_2, functions *slide_window()* and *handle_insert()*

```
    function handle_delete(ts, dtb, dt)
    { # (a) for every delete from db1
      # (b) either the data is also deleted from db2
      #     within 30h
      # (c) or within 30h earlier the data
      #     was inserted into db1
      # (d) and not migrated to db2 since,
      #     nor in the next 30h

        if (dtb == "db1")            # (a)
        {   found = 0;
            q = list_top(window);
            while (q.seq > 0)
            {   if (q.seq    == dt    # same data id
                && q.curly == 1       # into db1 (c)
                && ts − q.mark ≤ 108000) # ≤ 30 hr
                {   found = 1;
                    if (migrated_since(dt))  # (b)
                    {   Obligation[dt] = ts + 108000;
                    } else                   # (d)
                    {   Nomigration[dt] = ts + 108000;
                    }
                    break;
                }
                q = q.nxt;
            }
            if (!found)
            {       Obligation[dt] = ts + 108000; # (d)
            }
        } else  # db2
        {   if (Obligation[dt] != 0)
            {   if (ts > Obligation[dt]) # (b) > 30 hr
                {   print "data:␣" dt ":␣failed\n";
                }
                unset Obligation[dt];
    }   }   }
```

Fig. 17 Cobra checker script for property Del_1_2, function *handle_delete()*

and the corresponding element is deleted from the associative array *Nonmigration* that is used to keep track of these obligations.

Next, we look at the definition of function *handle_delete()*, which does most of the work in this case. It is shown in Fig. 17.

Processing is again fairly straightforward. We first check which database is the target of the *delete* event. If it is DB1, we check if the same data item was inserted into that database less than 30 hours ago by traversing the list of events that are remembered for this purpose. If the data item was indeed

```
function migrated_since(dt)
{   x = q;
    while (x.seq > 0)
    {   if (x.seq == dt        # data id matches
        &&  x.curly == 2)      # inserted into db2
        {       return 1;      # yes, it was migrated
        }
        x = x.nxt;
    }
    return 0;                  # not found
}
```

Fig. 18 Cobra checker script for property Del_1_2, function *migrated_since()*

inserted less than 30 hours ago, we have to create an obligation to check that it will not migrate to database DB2 within the next 30 hours or, if it has already been migrated, that the same data item will also be deleted from DB2 within the next 30 hours.

Since the matching data item is not necessarily at the head of the list, we traverse the list elements by following *.nxt* references in this case.

For deletions from DB2, we only have to check if an earlier obligation exists, and if so, delete it. This happens in the *else* clause of the conditional statement.

We see one more call to a function we have not defined yet in this case, which is function *migrated_since()*. Its definition is shown in Fig. 18.

In this case, the function returns a value, which is either one or zero. It simply searches the window of events starting from the point at which it was called, marked by token reference *q*. If a match of the data item is found, for an insert into database DB2 we know that the item was migrated and return one. If no match is found, zero is returned.

That completes the definition of the *delete* property.

5 Evaluation

All measurements were made on the same hardware platform: a 64-bit Intel 3.2 GHz 6-core system with 32 GB of memory running Windows 10, using Cygwin for the Cobra measurements and Ubuntu 16.04 under the Windows Subsystem for Linux (WSL) for the measurements with the Daut tool. We first look at some reference data that was provided in the original paper that first described the properties we have specified.

5.1 The MonPoly measurements

In the work that introduced the insert and delete properties and applied them to the Nokia log [6], only some performance measurements were given. The data did not include measurements for checking the full Nokia log. Table II in [6] described nine event fragments that were used for the measurements, each covering 24 hours worth of events, corresponding to 2.1% of the full log. A full verification of the properties, however, requires us to monitor at least 30 hours of events in the past and into the future, which our measurements allow. For our measurements, we did not use parallelizations, and we assume neither did the MonPoly measurements.

The system used for the measurements in [6] was given as a 1.15 GHz AMD quad-core computer (the operating system was not specified). This means that to compare with our measurements on a 3.2 GHz system, we should minimally divide their cpu-times by 3.2/1.15, corresponding to a speedup of about 2.78.

Table III in [6] gives runtimes and memory use for the nine tests performed for a number of properties using the authors' MonPoly tool. We are specifically interested in the results for

Table 2 MonPoly measurements for the Ins_1_2 and Del_1_2 properties, from [6] Table II and III. The normalized runtimes give the equivalent times on a 3.2 GHz system instead of the 1.15 GHz system that was used for the original measurements. The number of violations found was not reported

Log	Ins_1_2			Del_1_2		
	Memory (MB)	Runtime (s)	Normalized time (s)	Memory (MB)	Runtime (s)	Normalized time (s)
1	161	13,860	4,986	176	24	8.6
2	103	2,640	950	139	16	5.8
3	107	4,020	1,446	87	13	4.7
4	102	1,440	518	79	11	4.0
5	71	540	194	58	8	2.9
6	65	300	108	53	7	2.5
7	57	180	65	111	12	4.3
8	115	4,380	1,576	184	21	7.6
9	111	2,880	1,036	102	11	4.0
sum	892	30,240	10,228	989	123	41.6

Table 3 Number of events processed per second by the MonPoly reference tool

Property	Total events	Normalized time(s)	Events/second
Ins_1_2	8,209,334	10,228	803
Del_1_2	8,209,334	41.6	197,340

properties Ins_1_2 and Del_1_2 here. The MonPoly data are shown in Table 2. Times in [6] for the insert property were given in minutes and converted to seconds here.

By taking the number of events that are processed in the nine 24-hour fragments combined, which is given in [6] as 8,209,334, we can arrive at the event processing rate: the number of events processed per second for each property as a basis for comparison with the measurements with the Daut and Cobra tools. These results are shown in Table 3. Clearly, the processing rate for the delete property was significantly higher than for the insert property.

5.2 The Daut and Cobra measurements

The measurements with the Daut and Cobra tools compared with those of the MonPoly reference, are shown in Table 4. As noted, the measurements for both tools were made on the same desktop system, with Cobra running in text-only mode reading the input log from the standard input, with a stream buffer size of 500K bytes.

For both Cobra and Daut, the processing rate is notably higher than for the reference tool. Daut requires the least amount of memory to perform the verification of the Ins_1_2 property, using more than six times less memory. Cobra, on the other hand, uses less memory than Daut for the verification of the Del_1_2 property. Runtimes are comparable between Daut and Cobra, and especially close for the Del_1_2 property. We have no memory data for the MonPoly tool when applied to the full log, so we do not know how that tool would perform on this metric.

5.3 The specification effort and result

The Daut and Cobra monitors were constructed by the developers of the respective tools, that is, experts in the use of these tools, similar to the earlier experiments with the MonPoly tool. In spite of that it is fair to say that writing especially the Del_1_2 specification took some effort and time. This can be seen, for example, by the argumentation made for the correctness of the Daut monitor, which in fact was used to construct it. The resulting monitor specifications are clearly more verbose than the MonPoly versions. This is especially the case for the Del_1_2 property. The Daut monitors were developed starting from the MFOTL specifications, to ensure that the right monitors were implemented. The Cobra monitor was developed from the plain English formulation of the problem, using the MFOTL version only for clarification. The monitors were largely correct as constructed, except for a couple of errors, which were corrected, followed by some optimizations to improve performance further. In the case of the Cobra monitors this meant reducing the amount of logging information and related data that were used to debug the initial versions.

An interesting question is how these monitors compare to writing directly in a programming language without any support for trace analysis. The Cobra monitors are written in a style similar to how one might write such properties in, e.g., Python, using the Python dictionary and list data types. Writing the properties in C would require more effort due to the lack of built-in support for these data types.

6 Conclusion

In this case study, we compared a declarative logical specification formalism MFOTL, as used in the MonPoly tool, with two more operational specification methods, one based on an *internal* DSL and the other based on an *external* DSL. Perhaps not surprisingly, the more operational formalisms delivered the best performance, but required more writing to specify the target properties.

The Daut automata-based formalism is embedded into a programming language (Scala), which has the advantage that the user can reach outside the formalism to handle less common cases. It also adds the advantage of the compilation of the checks into optimized byte code, to improve the efficiency of the monitoring itself.

Table 4 Comparison of Memory Use and Event Processing Rates for MonPoly, Daut, and Cobra. The MonPoly runs processed a 9 day fragment of the log. The Daut and Cobra runs processed the entire log

	Ins_1_2				Del_1_2			
9 days	Mem (MB)	time (s)	events/sec	violations	Mem (MB)	time (s)	events/sec	violations
MonPoly	...	10,228	803	...	...	41.6	17,341	...
422.5 days	Mem (MB)	time (s)	events/sec	violations	Mem (MB)	time (s)	events/sec	violations
Daut	282	1,982	108,398	82,886	6,782	524	409,612	25
Cobra	1,746	575	373,467	82,886	2,375	473	453,800	25

The Cobra specification formalism, on the other hand, is based on a scripting language, implemented in C, that is normally used for specifying static analysis queries. The language is fairly simple, with support for recursive functions, the basic data types strings, integers, and references, and builtin support for lists, associative arrays, and hash-maps. Even though the scripts are interpreted with each script executed once for every token in the input stream, it achieved the shortest runtimes of the three methods we have considered here.

The use of the operational formalisms resulted in performance of one to two orders of magnitude greater than the logic-based formalism, more than sufficient to keep up with long-lasting event streams. In the case study, we considered an event log spanning 422.5 days, for instance, which could be processed in minutes. Considering the three E's of runtime verification: Elegance of notation, Expressiveness, and Efficiency, we can therefore see a decisive advantage for the operational approaches for the third E. Expressiveness is identical for both operational methods, since their respective languages are trivially Turing complete, which both are more expressive than the logical approach. That leaves Elegance, the first E. If conciseness matters, the advantage is clearly with the logic based method used in MonPoly, although, as in other fields, too much conciseness can in some cases compromise clarity and ease of use.

Acknowledgements We would like to thank members of the MonPoly team, David Basin, Matus Harvan, Felix Klaedtke, and Srdan Krstic for their responses to our questions. The research performed by the first author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The research of the second author was supported by Nimble Research, under a contract with Cyber Pack Ventures, Inc.

References

1. Aad, I., Niemi, V.: 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. Barringer, H., TraceContract, K.H.: A Scala DSL for trace analysis. In: Proc. 17th Int. Symposium on Formal Methods (FM 2011), Limerick, Ireland. LNCS, vol. 6664, pp. 57–72. Springer, Berlin (2011)
3. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: VMCAI. LNCS, vol. 2937, pp. 44–57. Springer, Berlin (2004)
4. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for runtime monitoring: from Eagle to RuleR. In: Proc. of the 7th Int. Workshop on Runtime Verification (RV'07), Vancouver, Canada. LNCS, vol. 4839, pp. 111–125. Springer, Berlin (2007)
5. Barringer, H., Havelund, K., Kurklu, E., Morris, R.: Checking flight rules with TraceContract: application of a Scala DSL for trace analysis. In: Scala Days 2011. Stanford University, California (2011)
6. Basin, D., Harvan, M., Klaedtke, F., Zalinescu, E.: Monitoring usage-control policies in distributed systems. In: Proc. of the 18th Int. Symp. on Temporal Representation and Reasoning, pp. 88–95 (2011)
7. Basin, D.A., Klaedtke, F., Marinovic, S., Zălinescu, E.: Monitoring of temporal first-order properties with aggregations. Form. Methods Syst. Des. **46**(3), 262–285 (2015)
8. Bauer, A., Leucker, M.: The theory and practice of SALT. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods. LNCS, vol. 6617, pp. 13–40. Springer, Berlin (2011)
9. Cobra on github. https://github.com/nimble-code/Cobra (2022)
10. Colombo, C., Pace, G.J., Schneider, G.: 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, pp. 33–37. IEEE Computer Society, Washington (2009)
11. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: Proceedings of TIME 2005: The 12th International Symposium on Temporal Representation and Reasoning, pp. 166–174. IEEE, New York (2005)
12. Daut on github. https://github.com/havelund/daut (2022)
13. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. Int. J. Softw. Tools Technol. Transf. **18**(2), 205–225 (2016)
14. FastCSV on github. https://github.com/osiegmar/FastCSV (2022)
15. Gorostiaga, F., HStriver, C.S.: A very functional extensible tool for the runtime verification of real-time event streams. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) Formal Methods. LNCS, vol. 13047, pp. 563–580. Springer, Berlin (2021)
16. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Trans. Serv. Comput. **5**(2), 192–206 (2012)
17. Havelund, K.: Data automata in Scala. In: 2014 Theoretical Aspects of Software Engineering Conference, TASE 2014, Changsha, China, September 1–3, 2014, pp. 1–9. IEEE Computer Society, Washington (2014)
18. Havelund, K.: Rule-based runtime verification revisited. Int. J. Softw. Tools Technol. Transf. **17**(2), 143–170 (2015)
19. Holzmann, G.J.: Cobra: a light-weight tool for static and dynamic program analysis. Innov. Syst. Softw. Eng. **13**(1), 35–49 (2017)
20. Joshi, R.: Personal communication (2019)
21. Kallwies, H., Leucker, M., Schmitz, M., Schulz, A., Thoma, D., Weiss, A.: TeSSLa – an ecosystem for runtime verification. In: Dang, T., Stolz, V. (eds.) Runtime Verification – 22nd International Conference, RV 2022, Tbilisi, Georgia, September 28–30, 2022. LNCS, vol. 13498, pp. 314–324. Springer, Berlin (2022)
22. Kim, M., Kannan, S., Lee, I., Sokolsky, O., Viswanathan, M.: Java-MaC: a run-time assurance tool for Java. Electron. Notes Theor. Comput. Sci. **55**(2), 218–235 (2001)
23. Kurklu, E., Havelund, K.: A flight rule checker for the LADEE Lunar spacecraft. In: 17th International Colloquium on Theoretical Aspects of Computing (ICTAC'20). LNCS, vol. 12545 (2020)
24. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. Int. J. Softw. Tech. Technol. Transf. **14**, 249–289 (2012). https://doi.org/10.1007/s10009-011-0198-6
25. Odersky, M., Spoon, L., Venners, B.: Programming in Scala: Updated for Scala 2.12, 3rd edn. Artima Incorporation, USA (2016)
26. Perez, I., Dedden, F., Goodloe, A.: Copilot 3. Technical Report NASA/TM-2020-220587, NASA Langley Research Center, (2020). https://ntrs.nasa.gov/citations/20200003164
27. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57. IEEE Computer Society, Washington (1977)

28. Reger, G., Cruz, H.C., MarQ, D.R.: Monitoring at runtime with QEA. In: Baier, C., Tinelli, C. (eds.) Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015), pp. 596–610. Springer, Berlin (2015)
29. Scala programming language. http://www.scala-lang.org (2022)
30. Shafiei, N., Havelund, K., Mehlitz, P.: Actor-based runtime verification with MESA. In: 20th International Conference on Runtime Verification (RV'20). LNCS, vol. 12399. Springer, Berlin (2020)
31. TraceContract on github. https://github.com/havelund/tracecontract (2022)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.