The DejaVu Runtime Verification Benchmark*

Klaus Havelund[1], Doron Peled[2], and Dogan Ulus[3]

[1] Jet Propulsion Laboratory, California Inst. of Technology, Pasadena, CA, USA
[2] Department of Computer Science, Bar Ilan University, Ramat Gan, Israel
[3] Boston University, Boston, MA, USA

1 Introduction

In this paper we present a benchmark for evaluating runtime verification tools. It was originally created in order to compare the DEJAVU runtime verification tool[1] with another similar tool. DEJAVU's logic is first-order past time temporal logic. In order to monitor such properties efficiently, Binary Decision Diagrams (BDDs) [1] are used for representing the data observed in a trace. The details on the logic and its algorithm are described in e.g. [2, 3, 4]. The benchmark consists of six properties, formulated in English, and formalized in DEJAVU's logic. For each property is provided (normally) three traces, of sizes varying from 10,000 events to one million events. Traces are represented in CSV format.

2 The DejaVu Logic

2.1 Syntax

The formulas of the core logic (minimal set of operators) are defined by the following grammar, where p is a predicate (an event), a is a constant, and x is a variable. For simplicity of the presentation, we define here the logic with unary predicates, but this is not due to any principle limitation, and, in fact, our implementation supports predicates with multiple arguments, including zero arguments, which correspond to propositions.

$$\varphi ::= \mathit{true} \mid p(a) \mid p(x) \mid (\varphi \wedge \varphi) \mid \neg\varphi \mid (\varphi \, S \, \varphi) \mid \ominus \varphi \mid \exists x \, \varphi$$

At a given state the formula $p(\text{“a”})$ means that $p(\text{“a”})$ happened, that is, $p(\text{“a”})$ is among the ground predicates of the state. Consider now the formula $p(x)$. We interpret it such that x is assigned any value “a” where $p(\text{“a”})$ appears in the current state. Thus, for interpreting $(p(x) \wedge q(y))$ in a state that has the predicates $p(\text{“a”})$ and $q(3)$, we have the assignment $[x \mapsto \text{“a”}, y \mapsto 3]$. The formula $(\varphi_1 \, S \, \varphi_2)$ (reads φ_1 *since* φ_2) means that φ_2 occurred in the past (including now) and since then (beyond that state) φ_1 has been true. The property $\ominus \varphi$ means that φ is true in the previous state. Existential quantification has the “obvious” meaning.

We can define the following derived operators as syntactic sugar using the operators defined in the above core: $\mathit{false} = \neg\mathit{true}$, $\forall x \, \varphi = \neg\exists x \, \neg\varphi$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $\mathbf{P} \, \varphi = (\mathit{true} \, S \, \varphi)$ (“previously”), $\mathbf{H} \, \varphi = \neg\mathbf{P} \, \neg\varphi$ (“always in the past” or “historically”), and $[\varphi_1, \varphi_2) = (\neg\varphi_2 \, S \, \varphi_1)$ (a semi-open interval).

*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 performed by the second author was partially funded by Israeli Science Foundation grant 2239/15: “Runtime Measuring and Checking of Cyber Physical Systems”.
© 2018. All rights reserved.

[1] The DEJAVU tool is available at https://github.com/havelund/dejavu.

2.2 Semantics

An *assignment* over a set of variables maps each variable x to a value. For example $[x \to 5, y \to \text{“abc”}]$ assigns the values 5 to x and the value “abc” to y. A *state* is a finite set of ground predicates, also referred to as *events*, where each predicate name may appear at most once. An *execution* $\sigma = s_1 s_2 \ldots$ (observed at any time) is a finite sequence of *states*.

Let $vars(\varphi)$ be the set of free (i.e., unquantified) variables of a subformula φ. We denote by $\gamma|_{vars(\varphi)}$ the restriction (projection) of an assignment γ to the free variables appearing in φ. Let ε be the empty assignment and let γ be an assignment to the variables that appear free in a formula φ. Then $(\gamma, \sigma, i) \models \varphi$ iff. φ holds for the prefix $s_1 s_2 \ldots s_i$ of the trace σ. In any of the following cases, $(\gamma, \sigma, i) \models \varphi$ is defined when γ is an assignment over $vars(\varphi)$, and $i \geq 1$.

- $(\varepsilon, \sigma, i) \models \mathit{true}$.
- $(\varepsilon, \sigma, i) \models p(a)$ if $p(a) \in \sigma[i]$.
- $([v \mapsto a], \sigma, i) \models p(v)$ if $p(a) \in \sigma[i]$.
- $(\gamma, \sigma, i) \models (\varphi \wedge \psi)$ if $(\gamma|_{vars(\varphi)}, \sigma, i) \models \varphi$ and $(\gamma|_{vars(\psi)}, \sigma, i) \models \psi$.
- $(\gamma, \sigma, i) \models \neg\varphi$ if not $(\gamma, \sigma, i) \models \varphi$.
- $(\gamma, \sigma, i) \models (\varphi \, S \, \psi)$ if for some $1 \leq j \leq i$, $(\gamma|_{vars(\psi)}, \sigma, j) \models \psi$ and for all $j < k \leq i$, $(\gamma|_{vars(\varphi)}, \sigma, k) \models \varphi$.
- $(\gamma, \sigma, i) \models \ominus\varphi$ if $i > 1$ and $(\gamma, \sigma, i-1) \models \varphi$.
- $(\gamma, \sigma, i) \models \exists x \, \varphi$ if there exists a value a such that[2] $(\gamma[x \mapsto a], \sigma, i) \models \varphi$.

3 Properties

We present six properties below. DEJAVU accepts properties in ASCII format. In the formulas below the used notation should be straightforward, although we point out that exclamation mark ‘!’ stands for $\neg$ (negation), ‘&’ stands for $\wedge$ (conjunction), and ‘@’ stands for $\ominus$ (previous state).

3.1 Access Property

3.1.1 Informally

> *The Access property states that if a file is accessed by a user, then the user should have logged in and not yet logged out, and the file should have been opened and not yet closed.*

3.1.2 Events

The events of interest are the following.

Event	Explanation
access(user,file)	user accesses file
login(user)	user logs in
logout(user)	user logs out
open(file)	file is opened
close(file)	file is closed

[2] $\gamma[x \mapsto a]$ is the overriding of γ with the binding $[x \mapsto a]$.

3.1.3 Formally

The formalization in DEJAVU's logic is as follows.

```
prop access :
  forall u . forall f . access(u,f) → [login(u), logout(u)) & [open(f), close(f))
```

3.1.4 Example Traces

Examples of satisfying and violating traces are:

- Satisfy : *login(John).open(file1).access(John,file1).close(file1).logout(John)*
- Violate : *login(John).open(file1).close(file1).access(John,file1).logout(John)*

3.2 File Property

3.2.1 Informally

> *The File property states that if a file is closed, then it must have been opened (and not yet closed) with some mode m (e.g. read or write).*

3.2.2 Events

The events of interest are the following.

Event	Explanation
open(file,mode)	file is opened with a particular mode (e.g, read or write)
close(file)	file is closed

3.2.3 Formally

The formalization in DEJAVU's logic is as follows.

```
prop file :
  forall f . close(f) → exists m . @ [open(f,m),close(f))
```

3.2.4 Example Traces

Examples of satisfying and violating traces are:

- Satisfy : *open(file2,write).close(file2)*
- Violate : *open(file1,write).close(file2)*

3.3 FIFO Property

3.3.1 Informally

The FIFO property is a conjunction of four subproperties about data entering and exiting a queue:

- *A datum can enter the queue at most once.*
- *A datum can exit the queue at most once.*
- *A datum can only exit if it has previously been entered.*
- *The queue has to respect the FIFO principle.*

3.3.2 Events

The events of interest are the following.

Event	Explanation
enter(x)	datum x is entered into the queue
exit(x)	datum x is removed from the queue

3.3.3 Formally

The formalization in DEJAVU's logic is as follows.

```
prop fifo :
  forall x .
    ( enter (x) → ! @ P enter (x)) &
    ( exit (x) → ! @ P exit (x)) &
    ( exit (x) → @ P enter (x)) &
    ( forall y . ( exit (y) & P (enter (y) & @ P enter (x))) →
      @ P exit (x))
```

3.3.4 Example Traces

Examples of satisfying and violating traces are:

- Satisfy : *enter(1).enter(2).exit(1).exit(2)*
- Violate : *enter(1).enter(2).exit(2).exit(1)*

3.4 Lock Property

3.4.1 Informally

This property concerns the acquisition and release of locks by concurrently executing threads. It consists of three subproperties:

- *A thread going to sleep must have released all acquired locks before then.*
- *If a thread acquires a lock, no thread may prior have acquired the lock and not yet released it.*
- *A thread cannot release a lock without having acquired it and not yet released it.*

3.4.2 Events

The events of interest are the following.

Event	Explanation
acq(thread,lock)	thread acquires lock
rel(thread,lock)	thread releases lock
sleep(thread)	thread goes to sleep

3.4.3 Formally

The formalization in DEJAVU's logic is as follows.

```
prop locking :
  forall t . forall l .
    (
      (sleep(t) → ![acq(t,l), rel(t,l))) &
      (acq(t,l) → ! exists s . @ [acq(s,l), rel(s,l))) &
      (rel(t,l) → @ [acq(t,l), rel(t,l)))
    )
```

3.4.4 Example Traces

Examples of satisfying and violating traces are:

- Satisfy : *acq(t,l).rel(t,l).sleep(t)*
- Violate : *acq(t,l).sleep(t).rel(t,l)*

3.5 Deadlock Property

3.5.1 Informally

The Deadlock property states that any two threads are not allowed to acquire any two locks in opposite order. That is, if a thread t_1 acquires a lock l_1, and then before releasing it, acquires a lock l_2, then another thread t_2 is not allowed to first acquire l_2 and then, before releasing it, acquire l_1. Following this discipline prevents cyclic deadlocks between two threads. Note that a violation of this property in a trace only indicates that the monitored application has a potential for deadlocking (in a different schedule), not that it is necessarily deadlocking in the observed schedule.

3.5.2 Events

The events of interest are the following.

Event	Explanation
acq(thread,lock)	thread acquires lock
rel(thread,lock)	thread releases lock

3.5.3 Formally

The formalization in DEJAVU's logic is as follows.

```
prop deadlock :
  forall t1 . forall t2 . forall l1 . forall l2 .
    (@ [acq(t1,l1), rel(t1,l1)) & acq(t1,l2)) →
      (! @ P (@ [acq(t2,l2), rel(t2,l2)) & acq(t2,l1)))
```

3.5.4 Example Traces

Examples of satisfying and violating traces are:

- Satisfy : *acq(t1,l1).rel(t1,l1).acq(t1,l2).rel(t2,l2).acq(t2,l2).rel(t2,l2).acq(t2,l1).rel(t2,l1)*
- Violate : *acq(t1,l1).acq(t1,l2).rel(t1,l2).rel(t1,l1).acq(t2,l2).acq(t2,l1).rel(t2,l1).rel(t2,l2)*

3.6 Datarace Property

3.6.1 Informally

> *The Datarace property captures data race potentials. A data race occurs when two threads access (read or write) the same shared variable simultaneously, and at least one of the threads writes to the variable. The property states that if a variable is acccessed by two threads, and one writes to the variable, there must exist a lock, which both threads hold whenever they access the variable.*

3.6.2 Events

The events of interest are the following.

Event	Explanation
acq(thread,lock)	thread acquires lock
rel(thread,lock)	thread releases lock
read(thread,var)	thread reads variable
write(thread,var)	thread writes variable

3.6.3 Formally

The formalization in DEJAVU's logic is as follows.

```
prop datarace :
  forall t1 . forall t2 . forall x .
    ((P (read(t1,x) | write(t1,x))) & (P write(t2,x))) →
      exists l .
        (H ((read(t1,x) | write(t1,x)) → [acq(t1,l), rel(t1,l))) &
         H ((read(t2,x) | write(t2,x)) → [acq(t2,l), rel(t2,l))))
```

3.6.4 Example Traces

Examples of satisfying and violating traces are:

- Satisfy : *acq(t1,l).write(t,x).rel(t1,l).acq(t2,l).read(t2,x).rel(t2,l)*
- Violate : *acq(t1,l1).write(t,x).rel(t1,l1).acq(t2,l2).read(t2,x).rel(t2,l2)*

4 Traces

4.1 Trace Format

The traces are represented in CSV (Comma Separated Value) format without headers. For example a trace consisting of the following five events:

login(John). open(file2). access(John,file2). close(file2). logout(John)

is represented as the following CSV file:

```
login,John
open,file2
access,John,file2
close,file2
logout,John
```

4.2 Trace Structure

For each property is provided (normally) three traces, of sizes approximately 10,000 events, 100,000 events, and one million events respectively. Traces have the general form that initially numerous opening events (login, open, enter, acq) occur, in order to accumulate a large amount of data stored in the monitor, after which a smaller number of corresponding closing events (logout, close, exit, rel) occur.

References

[1] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Survey*, 24:293–318, 1992.

[2] K. Havelund and D. Peled. Efficient runtime verification of first-order temporal properties. In *25th International Symposium on Model Checking of Software (SPIN 2018), 20-22 June, Malaga, Spain*, volume 10869 of *LNCS*. Springer, 2018.

[3] K. Havelund and D. Peled. Runtime verification: From propositional to first-order temporal logic (tutorial). In *18th International Conference on Runtime Verification (RV 2018), 10-13 November, Limassol, Cyprus.*, LNCS. Springer, 2018.

[4] K. Havelund, D. Peled, and D. Ulus. First-order temporal logic monitoring with BDDs. In *17th Conference on Formal Methods in Computer-Aided Design (FMCAD 2017), 2-6 October, Vienna, Austria*. IEEE, 2017.