

On-time diagnosis of discrete event systems: some examples

Aditya Mahjan and Demosthenis Teneketzis

June 4, 2008

Introduction

This article is a supplement to the material presented in the paper

“Aditya Mahajan and Demosthenis Teneketzis, *On-time diagnosis of discrete event systems*, Proceedings of the 9th International Workshop on Discrete Event Systems (WODES), pp. 382–387, Gothenberg, Sweden, May 28–30, 2008”

We present detailed computations of the example of Section IV of the paper and include the Haskell source code for computing the optimal on-time monitoring policy.

Preliminaries

```
{-# OPTIONS_GHC -XTypeSynonymInstances -XFlexibleInstances -XUnicodeSyntax #-}  
module Main where  
import Data.Tree  
import Data.Array ( (!), array, Array, )  
import Data.Ord (comparing)
```

To make things somewhat faster, we use the `Stream` library. The program can also be run without using streams, it consumes fractionally more heap memory.

```
import Prelude hiding ( (++) , map , length , maximum , replicate )  
import Data.List.Stream  
import Control.Monad.Stream
```

We first define some basic datatypes that we need for the rest of the program

```
data Obs    = Obs    Bool deriving (Eq, Show)  
data Term   = Term   Bool deriving (Eq, Show)  
data Alarm  = Alarm  Bool deriving (Eq, Show)
```

Language We restrict attention to a prefix-closed, finite and bounded language. The language is represented as a tree of events, where each event is a three-tuple denoting the event label, the observability of the event, and the cost incurred if the system stops immediately after executing the event. Thus¹

```
type UEvent    a = (String, Obs, Maybe a)  
type ULanguage a = Tree (UEvent a)
```

¹ The prefix `U` stands for `UnLabelled`

In the first step for obtaining the monitoring rule, we need to label each event. Each labelled event is a five-tuple consisting of the string, the projection of the string, a label to indicate whether the string is terminal or not, a label to indicate whether the last event of the string is observable or not, and the cost incurred if the system is stopped immediately after executing that string. Thus²

```
type LEvent    a = (String, String, Term, Obs, Maybe a)
type LLanguage a = Tree (LEvent a)
```

In the second step of obtaining the monitoring rule, we need an augmented observer. So, we define³

```
type OEvent    a = [LEvent a]
type OLanguage a = Tree (LEvent a)
```

Finally, we define an `Event` class and a data type `Language` to denote a generic event and a generic language.

```
data Language a = ULanguage a | LLanguage a | OLanguage a

class (Show a) => Event a where
  showLanguage  :: Tree a -> String
  printLanguage :: Tree a -> IO ()
  showEvent     :: a     -> String
  observation   :: a     -> String
  showLanguage  = drawTree . fmap showEvent
  printLanguage = putStr . showLanguage
  showEvent     = show
```

Now we define an instance of `Event` class for all events.

```
instance (Show a) => Event (UEvent a) where
  observation (_,Obs False,_) = ""
  observation (s,Obs True ,_) = s
```

```
instance (Show a) => Event (LEvent a) where
  observation (_,s,_,_,_) = s
```

```
instance (Show a) => Event (OEvent a) where
  observation [] = ""
  observation (x:xs) = observation x
```

We also need instances for `Event (String, Maybe a, Alarm)` and `Event (String, Maybe a, Maybe a)` for drawing the output of the performance tree and the optimal policy.

```
instance (Show a) => Event (String, Maybe a, Alarm) where
  observation (s,_,_) = s
```

```
instance (Show a) => Event (String, Maybe a, Maybe a) where
  observation (s,_,_) = s
```

² The prefix `L` stands for `Labelled`

³ The prefix `O` stands for `Observed`

The Optimal Monitoring Policy

As explained in the paper, the optimal monitoring policy can be obtained as follows. The description is slightly modified to conform with the code used here.

Step 1 (Labelling): Denote each string by a five tuple. The first component denotes the string, the second component denotes its projection, the third component indicates whether the string is terminal or not, the fourth components indicates if the last event of the string was observable or not, and the last component indicates the cost of cost of stopping at that string (if the system cannot stop at that string, the last component is `Nothing`).

This can be described as follows:

```
label :: ULanguage a -> LLanguage a
label = labelTree ("","")
```

`labelTree` and `labelForest` recursive label the sublanguage.

```
labelTree  :: (String, String) -> Tree  (UEvent a) -> Tree  (LEvent a)
labelForest :: (String, String) -> Forest (UEvent a) -> Forest (LEvent a)
```

```
labelTree (p,o) (Node (s, Obs True,  c) []) = let p' = p++s; o' = o++s in
      Node (p',o', Term True,  Obs  True, c) []
labelTree (p,o) (Node (s, Obs False, c) []) = let p' = p++s; o' = o      in
      Node (p',o', Term True,  Obs False, c) []
```

```
labelTree (p,o) (Node (s, Obs True,  c) t) = let p' = p++s; o' = o++s in
      Node (p',o', Term False, Obs  True, c)
      (labelForest (p',o') t)
```

```
labelTree (p,o) (Node (s, Obs False, c) t) = let p' = p++s; o' = o      in
      Node (p',o', Term False, Obs False, c)
      (labelForest (p',o') t)
```

```
labelForest prefix = map (labelTree prefix)
```

Step 2 (The augmented observer) Draw the augmented observer of the labelled language of step 1. The state of the observer after string t is observed contained all strings of the language belonging to the inverse projection $P^{-1}(t)$. The strings with `Obs` label constitute $Q(t)$ and the strings with `Term` label constitute $Q_T(t)$.

This can be done as follows:⁴

```
observer :: (Eq a, Show a) => Tree (LEvent a) -> Tree (OEvent a)
observer lang = fst $ until fixed next first where
  fixed (l1,l2) = l1 == l2
  next  (l1,l2) = shiftObserve l2
  first = shiftObserve $ fmap (:[]) lang
  shiftObserve l = (l, observeTree l)
```

`observeTree` and `observeForest` recursively obtain the observer of the sublanguage.

⁴ This recursive method of obtaining an observer is not too elegant. We need to figure out a one pass mechanism of constructing an observer.

```

observeTree :: (Show a) => Tree (OEvent a) -> Tree (OEvent a)
observeTree t@(Node x []) = t
observeTree (Node x f) = mergeTree (Node x diff) same where
  (same,diff) = let obs = observation x in
    partition (\(Node x' _) -> observation x' == obs) (observeForest f)

```

```

observeForest :: (Show a) => Forest (OEvent a) -> Forest (OEvent a)
observeForest [] = []
observeForest f = combineForest (map observeTree f)

```

The functions `mergeTree` and `combineForest` are defined in the Appendix.

Step 3 (The performance tree): For each observation in t calculate $(\max_{s \in Q(t)} C(s), \max_{s \in Q_T(t)} C(s))$. Notice that $\max_{s \in Q(t)} C(s)$ is the maximum cost of all the strings with label `Obs` in the observer state t . Similarly, $\max_{s \in Q_T(t)} C(s)$ is the maximum of all strings with label `Term` in the observer state t . If there are no strings with the label `Obs` or `Term`, we denote the corresponding maximum with a blank.

The performance tree can be computed as follows.

```

performance :: (Ord a, Show a) =>
  Tree (OEvent a) -> Tree (String, Maybe a, Maybe a)
performance = fmap (\x -> (observation x, maxQ x, maxQT x))

```

where

```

maxQ, maxQT :: (Ord a) => OEvent a -> Maybe a
maxQ = maxQby (\(_,_,_,Obs o,_) -> o)
maxQT = maxQby (\(_,_,Term t,_,_) -> t)

maxQby :: (Ord a) => (LEvent a -> Bool) -> OEvent a -> Maybe a
maxQby p xs = maximum $ map extract xs where
  extract x@(_,_,_,_,c) | p x = c
  | otherwise = Nothing

```

Step 4 (The optimal monitoring rule): The performance tree of step 3 contains all the information needed to evaluate the algorithm of Theorem 1 of the paper. We can start with the leaf nodes of the tree and compute the optimal monitoring policy by moving up the tree.

This can be computed as follows

```

monitorTree :: (Num a, Ord a, Show a) =>
  Tree (String, Maybe a, Maybe a) -> Tree (String, Maybe a, Alarm)
monitorTree (Node x@(s,c1,c2) [])
  | c1 `smaller` c2 = Node (s,c1,Alarm True) []
  | otherwise = Node (s,c2,Alarm False) []

monitorTree (Node x@(s,c1,c2) f)
  | c1 `smaller` c3 = Node (s,c1, Alarm True) f'
  | otherwise = Node (s,c3, Alarm False) f'
  where c3 = max c2 (cost worst)
        f' = monitorForest f
        worst = maximumBy (comparing cost) f'

```

```
cost (Node (_,c,_ _) = c
```

```
monitorForest :: (Num a, Ord a, Show a) =>  
  Forest (String, Maybe a, Maybe a) -> Forest (String, Maybe a, Alarm)  
monitorForest = map monitorTree
```

The function `smaller` is defined in the appendix.

The optimal policy This function obtains the optimal policy of a instance of costs.

```
policy :: Array Int Int -> Tree (String, Maybe Int, Alarm)  
policy = monitorTree . performance . observer . label . language
```

Displaying Results We also define a diagnostic function, which prints the intermediate results.

```
showPolicy :: Array Int Int -> IO ()  
showPolicy costs = do  
  let l      = language costs  
      step1  = label l  
      step2  = observer step1  
      step3  = performance step2  
      step4  = monitorTree step3  
      showResult "Step 1 (Labelling)"          step1  
      showResult "Step 2 (Augmented observer)" step2  
      showResult "Step 3 (The performance tree)" step3  
      showResult "Step 4 (Optimal monitoring rule)" step4
```

```
showResult :: (Event a) => String -> Tree a -> IO ()  
showResult str lang = do  
  putStrLn str  
  putStrLn $ replicate (length str) '-'  
  putStrLn ""  
  printLanguage lang  
  putStrLn ""
```

```
showInstance :: String -> Array Int Int -> IO ()  
showInstance str ins = do  
  putStrLn ['\f']  
  putStrLn str  
  putStrLn $ replicate (length str) '-'  
  putStrLn ""  
  showPolicy ins  
  putStrLn ""
```

Instance 0 This is just to show the costs.

```
instance0 :: Array Int String  
instance0 = array (1,11) [(x,"C" ++ show x) | x <- [1..11]]
```

Example of the paper

Language The language of the paper is given by⁵

```
language :: Array Int a -> ULanguage a
language c =
  Node (".", Obs False, Nothing)
    [Node ("f", Obs False, Nothing)
      [Node ("a", Obs True, Just $ c!1)
        [Node ("b", Obs False, Nothing)
          [Node ("a", Obs True, Just $ c!3)
            [Node ("c", Obs True, Just $ c!7)
              [Node ("d", Obs False, Just $ c!11)
                []]]]
          ,Node ("d", Obs False, Nothing)
            [Node ("a", Obs True, Just $ c!4)
              [Node ("a", Obs True, Just $ c!8) []
                ,Node ("d", Obs False, Just $ c!5) []]]]]
      ,Node ("a", Obs True, Just $ c!2)
        [Node ("b", Obs False, Nothing)
          [Node ("a", Obs True, Just $ c!6)
            [Node ("a", Obs True, Just $ c!9)
              [Node ("a", Obs True, Just $ c!10) []]]]]]]]
```

Instance 1 (High false alarm penalty) When the false alarm penalty is high, the monitor will not raise an alarm until it is sure that a fault has occurred. To see this, consider the costs

```
instance1 :: Array Int Int
instance1 = array (1,11)
              [(2,10), (6,10), (9,10), (10,10),
               (1,1), (3,3), (4,3), (5,4), (7,4), (8,4), (11,5)]
```

Instance 2 (High penalty of continuing with a fault) When the penalty of continuing with a fault is high, the monitor will raise an alarm as soon as it suspects a fault. To see this consider the costs

```
instance2 :: Array Int Int
instance2 = array (1,11)
              [(2,1), (6,1), (9,1), (10,1),
               (1,10), (3,11), (4,11),
               (5,12), (7,12), (8,12), (11,13)]
```

Instance 3 (Catastrophic Event) When there is a possibility of a catastrophic event, the monitor should shut down the system before the event can take place. To see this consider the costs

```
instance3 :: Array Int Int
instance3 = array (1,11)
              [(2,10), (6,10), (9,10), (10,15),
               (1,1), (3,2), (4,4), (5,100),
               (7,3), (11,12), (8,12)]
```

⁵ Haskell does not seem to like UTF ϵ , so I am using $.$ to represent the ϵ event.

The main program

```
main = do
  putStrLn "Language L:"
  putStrLn "======"
  putStrLn ""
  printLanguage $ language instance0
  showInstance "Instance 1" instance1
  showInstance "Instance 2" instance2
  showInstance "Instance 3" instance3
```

Appendix

Here we define the functions `mergeTree` and `combineForest` used in Step 2. The `mergeTree` function takes a tree and a forest, and augments the `rootLabel` of the tree with the `rootLabel` of all the nodes in the forest, and adds the `subForest` of all the nodes of the forest as a `subForest` of the root node of the tree. (It is easier to read the code than the description).

```
mergeTree :: Tree (OEvent a) -> Forest (OEvent a) -> Tree (OEvent a)
mergeTree t [] = t
mergeTree (Node x f) (Node x' f':fs) = mergeTree (Node (x++x') (f++f')) fs
```

The `combineForest` function combines all the `subForest` of all the nodes that have the same observation.

```
combineForest :: (Show a) => Forest (OEvent a) -> Forest (OEvent a)
combineForest f = do_combineForest f []

do_combineForest :: (Show a) =>
  Forest (OEvent a) -> Forest (OEvent a) -> Forest (OEvent a)
do_combineForest [] ys = ys
do_combineForest (x:xs) ys = do_combineForest diff (ys ++ [mergeTree x same])
  where (same,diff) = let obs = observation $ rootLabel x in
    partition (\(Node x' _) -> observation x' == obs) xs
```

Now we define the function `smaller` used in step 4. This is basically comparison function for `Maybe` datatype

```
smaller Nothing _ = False
smaller _ Nothing = False
smaller (Just a) (Just b) = a < b
```

Result

Language L

```
(".",Obs False,Nothing)
|
+- ("f",Obs False,Nothing)
| |
|  `- ("a",Obs True,Just "C1")
|    |
|    +- ("b",Obs False,Nothing)
|      | |
|      |  `- ("a",Obs True,Just "C3")
|      |    |
|      |    `- ("c",Obs True,Just "C7")
|      |      |
|      |      `- ("d",Obs False,Just "C11")
|      |
|      `- ("d",Obs False,Nothing)
|        |
|        `- ("a",Obs True,Just "C4")
|          |
|          +- ("a",Obs True,Just "C8")
|            |
|            `- ("d",Obs False,Just "C5")
|
`- ("a",Obs True,Just "C2")
  |
  `- ("b",Obs False,Nothing)
    |
    `- ("a",Obs True,Just "C6")
      |
      `- ("a",Obs True,Just "C9")
        |
        `- ("a",Obs True,Just "C10")
```

Instance 1

Step 1 (Labelling)

```
(".", "", Term False, Obs False, Nothing)
|
+- (.f", "", Term False, Obs False, Nothing)
| |
| `-. (.fa", "a", Term False, Obs True, Just 1)
|   |
|   +- (.fab", "a", Term False, Obs False, Nothing)
|     | |
|     | `-. (.faba", "aa", Term False, Obs True, Just 3)
|       | |
|       | `-. (.fabac", "aac", Term False, Obs True, Just 4)
|         | |
|         | `-. (.fabacd", "aac", Term True, Obs False, Just 5)
|         |
|         `-. (.fad", "a", Term False, Obs False, Nothing)
|           |
|           `-. (.fada", "aa", Term False, Obs True, Just 3)
|             |
|             +- (.fadaa", "aaa", Term True, Obs True, Just 4)
|               |
|               `-. (.fadad", "aa", Term True, Obs False, Just 4)
|
|
+- (.a", "a", Term False, Obs True, Just 10)
|
| `-. (.ab", "a", Term False, Obs False, Nothing)
|   |
|   `-. (.aba", "aa", Term False, Obs True, Just 10)
|     |
|     `-. (.abaa", "aaa", Term False, Obs True, Just 10)
|       |
|       `-. (.abaaa", "aaaa", Term True, Obs True, Just 10)
```

Step 2 (Augmented observer)

```
[(".", "", Term False, Obs False, Nothing), (".f", "", Term False, Obs False, Nothing)]
|
`- [(".a", "a", Term False, Obs True, Just 10), (".ab", "a", Term False, Obs False, Nothing),
    (".fa", "a", Term False, Obs True, Just 1), (".fab", "a", Term False, Obs False, Nothing),
    (".fad", "a", Term False, Obs False, Nothing)]
|
`- [(".aba", "aa", Term False, Obs True, Just 10),
    (".faba", "aa", Term False, Obs True, Just 3),
    (".fada", "aa", Term False, Obs True, Just 3),
    (".fadad", "aa", Term True, Obs False, Just 4)]
|
+- [(".abaa", "aaa", Term False, Obs True, Just 10),
    (".fadaa", "aaa", Term True, Obs True, Just 4)]
| |
| `- [(".abaaa", "aaaa", Term True, Obs True, Just 10)]
|
`- [(".fabac", "aac", Term False, Obs True, Just 4),
    (".fabacd", "aac", Term True, Obs False, Just 5)]
```

Step 3 (The performance tree)

```
("", Nothing, Nothing)
|
`- ("a", Just 10, Nothing)
|
`- ("aa", Just 10, Just 4)
|
+- ("aaa", Just 10, Just 4)
| |
| `- ("aaaa", Just 10, Just 10)
|
`- ("aac", Just 4, Just 5)
```

Step 4 (Optimal monitoring rule)

```
("", Just 10, Alarm False)
|
`- ("a", Just 10, Alarm False)
|
`- ("aa", Just 10, Alarm False)
|
+- ("aaa", Just 10, Alarm False)
| |
| `- ("aaaa", Just 10, Alarm False)
|
`- ("aac", Just 4, Alarm True)
```

Instance 2

Step 1 (Labelling)

```
(".", "", Term False, Obs False, Nothing)
|
+- (.f", "", Term False, Obs False, Nothing)
| |
| `-. (.fa", "a", Term False, Obs True, Just 10)
|   |
|   +- (.fab", "a", Term False, Obs False, Nothing)
|     | |
|     | `-. (.faba", "aa", Term False, Obs True, Just 11)
|     |   |
|     |   `-. (.fabac", "aac", Term False, Obs True, Just 12)
|     |     |
|     |     `-. (.fabacd", "aac", Term True, Obs False, Just 13)
|     |
|     `-. (.fad", "a", Term False, Obs False, Nothing)
|       |
|       `-. (.fada", "aa", Term False, Obs True, Just 11)
|         |
|         +- (.fadaa", "aaa", Term True, Obs True, Just 12)
|         |
|         `-. (.fadad", "aa", Term True, Obs False, Just 12)
|
`-. (.a", "a", Term False, Obs True, Just 1)
   |
   `-. (.ab", "a", Term False, Obs False, Nothing)
       |
       `-. (.aba", "aa", Term False, Obs True, Just 1)
           |
           `-. (.abaa", "aaa", Term False, Obs True, Just 1)
               |
               `-. (.abaaa", "aaaa", Term True, Obs True, Just 1)
```

Step 2 (Augmented observer)

```
[(".", "", Term False, Obs False, Nothing), (".f", "", Term False, Obs False, Nothing)]
|
`- [(".a", "a", Term False, Obs True, Just 1), (".ab", "a", Term False, Obs False, Nothing),
    (".fa", "a", Term False, Obs True, Just 10), (".fab", "a", Term False, Obs False, Nothing),
    (".fad", "a", Term False, Obs False, Nothing)]
|
`- [(".aba", "aa", Term False, Obs True, Just 1),
    (".faba", "aa", Term False, Obs True, Just 11),
    (".fada", "aa", Term False, Obs True, Just 11),
    (".fadad", "aa", Term True, Obs False, Just 12)]
|
+- [(".abaa", "aaa", Term False, Obs True, Just 1),
    (".fadaa", "aaa", Term True, Obs True, Just 12)]
| |
| `- [(".abaaa", "aaaa", Term True, Obs True, Just 1)]
|
`- [(".fabac", "aac", Term False, Obs True, Just 12),
    (".fabacd", "aac", Term True, Obs False, Just 13)]
```

Step 3 (The performance tree)

```
("", Nothing, Nothing)
|
`- ("a", Just 10, Nothing)
|
`- ("aa", Just 11, Just 12)
|
+- ("aaa", Just 12, Just 12)
| |
| `- ("aaaa", Just 1, Just 1)
|
`- ("aac", Just 12, Just 13)
```

Step 4 (Optimal monitoring rule)

```
("", Just 10, Alarm False)
|
`- ("a", Just 10, Alarm True)
|
`- ("aa", Just 11, Alarm True)
|
+- ("aaa", Just 12, Alarm False)
| |
| `- ("aaaa", Just 1, Alarm False)
|
`- ("aac", Just 12, Alarm True)
```

Instance 3

Step 1 (Labelling)

```
(".", "", Term False, Obs False, Nothing)
|
+- (.f", "", Term False, Obs False, Nothing)
| |
| `-. (.fa", "a", Term False, Obs True, Just 1)
|   |
|   +- (.fab", "a", Term False, Obs False, Nothing)
|     | |
|     | `-. (.faba", "aa", Term False, Obs True, Just 2)
|       | |
|       | `-. (.fabac", "aac", Term False, Obs True, Just 3)
|         | |
|         | `-. (.fabacd", "aac", Term True, Obs False, Just 12)
|           |
|           `-. (.fad", "a", Term False, Obs False, Nothing)
|             |
|             `-. (.fada", "aa", Term False, Obs True, Just 4)
|               |
|               +- (.fadaa", "aaa", Term True, Obs True, Just 12)
|                 |
|                 `-. (.fadad", "aa", Term True, Obs False, Just 100)
|
| `-. (.a", "a", Term False, Obs True, Just 10)
|   |
|   `-. (.ab", "a", Term False, Obs False, Nothing)
|     |
|     `-. (.aba", "aa", Term False, Obs True, Just 10)
|       |
|       `-. (.abaa", "aaa", Term False, Obs True, Just 10)
|         |
|         `-. (.abaaa", "aaaa", Term True, Obs True, Just 15)
```

Step 2 (Augmented observer)

```
[(".", "", Term False, Obs False, Nothing), (".f", "", Term False, Obs False, Nothing)]
|
^- [(".a", "a", Term False, Obs True, Just 10), (".ab", "a", Term False, Obs False, Nothing),
    (".fa", "a", Term False, Obs True, Just 1), (".fab", "a", Term False, Obs False, Nothing),
    (".fad", "a", Term False, Obs False, Nothing)]
|
^- [(".aba", "aa", Term False, Obs True, Just 10),
    (".faba", "aa", Term False, Obs True, Just 2),
    (".fada", "aa", Term False, Obs True, Just 4),
    (".fadad", "aa", Term True, Obs False, Just 100)]
|
+- [(".abaa", "aaa", Term False, Obs True, Just 10),
    (".fadaa", "aaa", Term True, Obs True, Just 12)]
| |
| ^- [(".abaaa", "aaaa", Term True, Obs True, Just 15)]
|
^- [(".fabac", "aac", Term False, Obs True, Just 3),
    (".fabacd", "aac", Term True, Obs False, Just 12)]
```

Step 3 (The performance tree)

```
("", Nothing, Nothing)
|
^- ("a", Just 10, Nothing)
|
^- ("aa", Just 10, Just 100)
|
+- ("aaa", Just 12, Just 12)
| |
| ^- ("aaaa", Just 15, Just 15)
|
^- ("aac", Just 3, Just 12)
```

Step 4 (Optimal monitoring rule)

```
("", Just 10, Alarm False)
|
^- ("a", Just 10, Alarm False)
|
^- ("aa", Just 10, Alarm True)
|
+- ("aaa", Just 12, Alarm True)
| |
| ^- ("aaaa", Just 15, Alarm False)
|
^- ("aac", Just 3, Alarm True)
```