Bidirectional Transformations in Specifications for Runtime Verification

Lisandra Silva

National Institute of Informatics, Tokyo Japan

Abstract. There are a few different specifications languages to specify monitors for Runtime Verification, and reason about the same property in different specification languages is not straightforward. The present work uses Bidirectional Transformations to synchronize different specifications for Runtime Verification. More specifically to synchronize a specification expressed as a Regular Expression and another expressed as a Finite State Machine.

Keywords: Bidirectional Transformations · Runtime Verification · Regular Expressions · Non-deterministic Finite Automata · Deterministic Finite Automata

1 Introduction

Runtime Verification (RV) is a lightweight formal method to ensure, at execution time, that a system meets a desirable behaviour. A possible approach for RV consists in analyzing an execution trace of the system under scrutiny using a decision procedure called *monitor*. The desirable behaviour of the system can be specified as a set of properties to be verified. From each property a monitor is generated, whose primary goal is to detect violation or satisfaction with respect to the given specification, emitting verdicts (truth values) that indicate satisfaction or violation of the property [1, 4, 5].

There a few different specifications languages to specify monitors for RV, and comparing the expressiveness is these languages is not straightforward. Besides, sometimes it is not easy to select a specific language to write a specification, because the syntax and operations of a particular language may make certain specifications easier or harder to write and/or read [1, 5].

Some work has been done in translating between languages. For instance the translation of first-order temporal logic into quantified event automata. However, the present work will focus on the translation between Regular Expressions (RE) and Finite State Machine (FSM), two different specification languages to write RV properties.

There are several algorithms to convert between RE and FSM, and they will be summarized in the present work. However, the algorithms to go from RE to FSM and from FSM to RE are not inverse of each others. This means that given a RE we can obtain its corresponding FSM, but going back to RE can produce a completely different RE.

When reasoning about the translation of properties between different language specifications it would be useful that small changes in one resulted in small changes in the another. For a simple example, given the RE (ab*|ba*), when editing just one of the branches of the RE in the corresponding FSM, when coming back again to the RE only that branch should have been updated.

"The art of progress is to preserve order amid change and to preserve change amid order"

A N Whitehead

This is the main motivation to use Bidirectional Transformations (BX) to translate between RE and FSM. BX provide a mechanism to maintain consistency between two pieces of data, the *source* and the *view*. The *get* function extracts part of the information from the source and produces a view, while the *put* function takes the original source and a (possibly) updated view and produces the updated source. The BX must satisfy well-behavedness rules such as *putting* an unmodified view into a source must produce the original source. But besides that, we want to explore the least-changes principle, which states that "small" changes on the view lead to "small" changes on the source, a topic which is still unsettled and actively investigated by the BX community.

2 Runtime Verification

Runtime Verification (RV) is a dynamic analysis method aiming at checking whether a run of the system under scrutiny satisfies a given correctness property.

The components of a RV environment are the system to be checked and a set of properties to be checked against the system execution. Properties can be expressed in a formal specification language, or even as a program in a general-purpose programming language. From a given property, a monitor is generated, i.e., a decision procedure for that property - *monitor synthesis*. The system is instrumented to generate the relevant events to be fed into the monitor - *system instrumentation*. In the next step, the system's execution is analyzed by the monitor - *execution analysis*. The monitor is able to consume the events produced by the running system and, for each consumed event, emits a verdict indicating the status of the property, depending on the event sequence seen so far. Finally, the monitor sends feedback to the system so that more specific corrective actions can be taken (Figure 1) [5].

2.1 Formal Specification of the system behaviour

There are different formal approaches to describe the expected behaviour of a system. But before presenting some different specification languages, let's start by presenting some general properties of these formalisms.

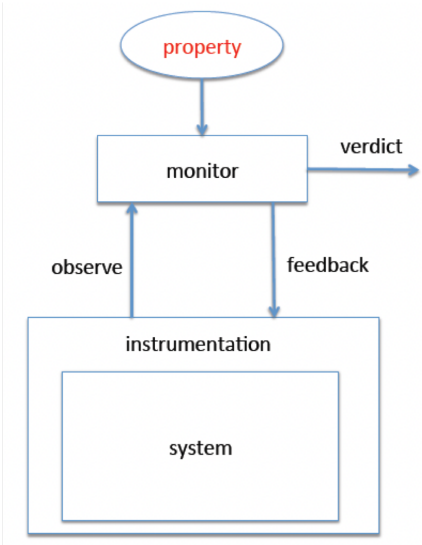


Fig. 1. An overview of the RV process

Events The behaviour of a system can be analyzed as the way the system changes over time, and this can be done through its observation. To abstract these observations we will use *Events* - discrete atomic entities that represent *actions* or *state changes* made by the system. The system's observable events of interest is called its *alphabet*. The choice of events is part of the specification and will determine the available information about the system and about which properties can be described.

Traces A *trace* is a sequence of events and abstracts the behaviour of a single run of the system. Obviously, an observable trace must be finite, but it is sometimes useful to think about the possible infinite behaviours of a system. Therefore, a trace can be viewed as a finite prefix of the infinite behaviour.

Properties and Specifications The abstraction of a property can be described as a set of traces and its specification is a concrete (textual) object that denotes this set of traces. As there are many specifications languages, one can have many specifications for a single property, but a property is unique and independent of the specification language. If the specification language is ambiguous, then the specific property may not be clear. Dealing with such ambiguities is a common issue in the specification process [1].

2.2 Different language specifications

This section presents a brief description of the main specification languages for RV, as well as some of their general features.

A RV specification language can be *Executable* if the specification is directly executable and therefore more low-level, or *Declarative* in which an executable object (monitor) is generated from the specification. Also, some specification languages are more suited to specify sets of finite traces whereas others are more suited to specify infinite traces. A specification may also capture *good* or *bad* behaviour. A match against a good behaviour specification represents *validation* of the desired property, whereas a match a bad behaviour specification represents *violation* of that property [1].

Regular Expressions Regular Expressions are a commonly used formalism in Computer Science for describing sets of strings, but can also be used in RV to describe traces of events, where the atoms are not characters but events. The regular expression matches if any *suffix* of the trace matches the expression - *suffix-matching* - to attest the validation or violation of the property (e.g., in the work of TraceMatches) [5].

Finite State Machines Finite State machines have the same expressive power of Regular Expressions but have the advantage of being directly executable, in opposition to regular expressions and temporal logic, since both require *monitor synthesis* to produce a monitor, which is usually described as some form of state machine.

The formalisms of **Linear Temporal Logic** and **Context Free Grammars** can also be used for specifications in RV, but since they are not the main focus of the present work they will not be further approached.

3 Bidirectional Transformations

Bidirectional Transformations (BX) provide a mechanism for maintaining consistency between two pieces of data. A bidirectional transformation consists of a pair of functions: a *get* function that may discard part of information from a source to produce a view, and a *put* function that accepts a source and a possible modified view, and reflects the changes in the source producing an updated source that is consistent with that view. The pair of functions should be *well-behaved* satisfying the next laws:

$$put\ s\ (get\ s) = s \quad (GetPut)$$
$$get\ (put\ s\ v)\ = v \quad (PutGet)$$

The *GetPut* law requires that if no changes are made on the view *get s* then the *put* function should produce the same unmodified source s. The *PutGet* law says that all the changes in the view should be reflected in the source, as so getting from an updated source computed by *put*ting a view v should retrieve the same v [7, 8].

The straightforward approach to write a bidirectional transformation is to write two separate functions in any language and show that they satisfy the *well-behavedness* rules. Although this ad hoc solution provides full control over both *get* and *put* transformations, verifying that they are *well-behaved* requires intricate reasoning about their behaviours. Besides, any modification to one of the transformations requires a redefinition of the other transformation as well as a new *well-behavedness* proof [7, 2].

A better alternative is to write a single program where both transformations can be described at the same time, using for that a *bidirectional programming language*. Many bidirectional languages have been proposed during the past years and the design challenge for all of them lies in striking a balance between expressiveness and reliability.

There are two approaches when designing bidirectional programming languages:

- *get-based* - allows the programmer to write the *get* function, deriving a suitable *putback* function. The problem with this approach is that the *get* function may not be injective and so there may exist many possible put functions that can be combined with it to form a valid BX.
- *putback-based* - allows the programmer to write the *put* from which the only get function is automatically derived. The resulting pair of functions must form a *well-behaved* bidirectional transformation.

Despite this, as the present work is still exploratory, it focused on studying the possibility of implement a BX between RE and DFA with the first alternative, where the two functions *get* and *put* are written separately.

4 Regular Expressions and Finite State Machine

Regular expressions and Finite Automata (FA) represent a valuable concept in theoretical computer science. Their equivalence is well known dating back to Kleene's paper in 1956. REs are well suited for human users and therefore often used as interfaces to specify certain patterns or languages, while automata immediately translate to efficient data structures and are suited for programming tasks. Obviously this fact raises the interest in conversion algorithms between them [6].

Moreover, when reasoning about the specifications of properties for RV, it can be useful to convert between them and observe how changes in one of the specifications are reflected in the other. This was the main motivation for use Bidirectional Transformations between these two types of specifications.

The next sections will describe some formal definitions of RE and FA as well as some of the well known algorithms to convert between them.

4.1 Formal definitions

Before introducing the conversion algorithms it is useful to present and formally describe the two types of automata.

Regular Expressions Given a vocabulary Σ, a symbol $a \in \Sigma$ and the regular expressions s and t, then RE can be defined inductively as following:

$$RE = \quad \emptyset \quad | \quad \epsilon \quad | \quad a \quad | \quad (s+t) \quad | \quad (s \ . \ t) \quad | \quad (s)^*$$

The language defined by a regular expression r, denoted by $L(r)$, is defined as follows:

$$\begin{aligned} L(\emptyset) &= \emptyset \\ L(\epsilon) &= \{\epsilon\} \\ L(a) &= \{a\} \\ L(s+t) &= L(s) \cup L(t) \\ L(s \ . \ t) &= L(s) \ . \ L(t) \\ L(s^*) &= L(s)^* \end{aligned}$$

Non-Deterministic Finite Automata (NDFA) Formally, a NDFA is described as follows:

$$A = (Q, \Sigma, q_0, F, \delta)$$

Where Q is a finite set of states, Σ is a finite set of input symbols (the vocabulary), $q_0 \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of accepting (final) states, and $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ is the *transition function*.

As the name suggests, the NDFA can transit to one or more states when consuming a given symbol. The next state is computed as the set of all possible states to which is possible to transit. The NDFA can have *ϵ-transitions*, transitions without consuming any input symbol, i.e. the empty string ϵ is a possible input. If the NDFA has no ϵ-transitions, i.e. is ϵ-free, then the transition function can be restricted to $\delta : Q \times \Sigma \to 2^Q$.

Deterministic Finite Automata (DFA) *Deterministic* refers to the uniqueness of the computation. It can be seen as a special kind of NDFA in which for each state and symbol, the result of the transition has exactly one state. Following is the formal definition of a DFA:

$$A = (Q, \Sigma, q_0, F, \delta)$$

Where Q is the finite set of states, Σ is the finite set of input symbols (the vocabulary), $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting (final) states, and $\delta : Q \times \Sigma \to Q$ is the *transition function*.

4.2 Conversion algorithms

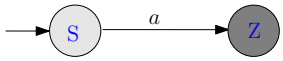
This sections review two of the most popular algorithms for conversion from a RE to a finite state automata.

Thompson's construction algorithm Converts a RE into an equivalent NDFA. The algorithm splits an expression into its constituent sub-expressions and works recursively applying the following rules:

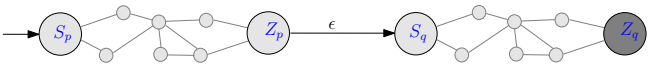
- $\mathbf{e} = \emptyset$ is converted in:

- $\mathbf{e} = \epsilon$ is converted in:

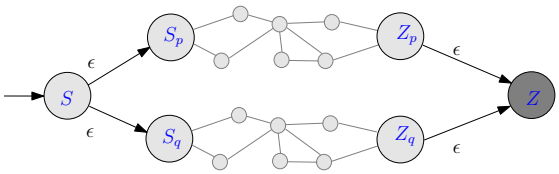
- $\mathbf{e} = \mathbf{a}$ is converted in:



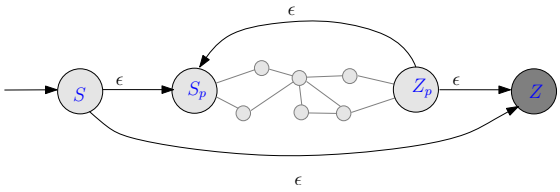
- $\mathbf{e} = \mathbf{p} \cdot \mathbf{q}$ is converted in:



- $\mathbf{e} = \mathbf{p} + \mathbf{q}$ is converted in:



- $\mathbf{e} = \mathbf{p}^*$ is converted in:



For instance, the regular expression $\mathbf{e} = (\epsilon + \mathbf{a}^*\mathbf{b})$ would be converted in the NDFA presented in Figure 2

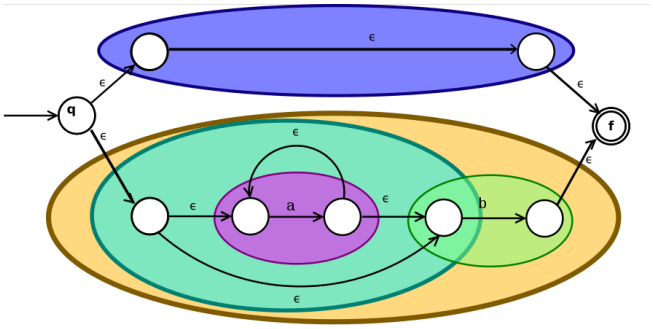


Fig. 2. NDFA resulting from $\mathbf{e} = (\epsilon + \mathbf{a}^*\mathbf{b})$ using Thompson's construction

Glushkov's construction algorithm Is another well-known algorithm to convert a give RE into a NDFA. It is similar to Thompson's construction, once the ϵ-transitions are removed. Following are the construction steps to create a NDFA that accepts the language $L(e)$ accepted by the RE e:

- **Step 1** - linearisation of the expression. Each letter of the alphabet appearing in the expression e is renamed, so that each letter occurs at most once in the new expression e'. Let A be the old alphabet and let B be the new one;
- **Step 2a** - the following sets are computed:
  - $P(e') = \{\ x \in B \quad | \quad xB^* \cap L(e') \neq \emptyset\ \}$ is the set of symbols which occur as a first letter of a word of $L(e')$. It can be defined inductively as:

$$\begin{aligned} P(\emptyset) = P(\epsilon) &= \emptyset \\ P(a) &= \{a\}, \text{ for each letter } a \\ P(s+t) &= P(s) \cup P(t) \\ P(s\ .\ t) &= P(s)\ \cup\ \Lambda(s)P(t) \\ P(s^*) &= P(s) \end{aligned}$$

- $D(e') = \{\ y \in B \quad | \quad B^*y \cap L(e') \neq \emptyset\ \}$ is the set of symbols that can end a word of $L(e')$. It can be defined inductively with the same rules of P except for the product, in which case,

$$D(s\ .\ t) = D(t)\ \cup\ D(s)\Lambda(t)$$

- $F(e') = \{\ u \in B^2 \quad | \quad B^*uB^* \cap L(e') \neq \emptyset\ \}$ is the set f symbol pairs that can occur in words of $L(e')$. It can be defined inductively as follows:

$$\begin{aligned} F(\emptyset) = F(\epsilon) = F(a) &= \emptyset, \text{ for each letter } a \\ F(s+t) &= F(s) \cup F(t) \\ F(s\ .\ t) &= F(s)\ \cup\ F(t)\ \cup\ D(s)P(t) \\ F(s^*) &= F(s)\ \cup\ D(e)P(e) \end{aligned}$$

– **Step 2b** - computes the set $\Lambda = \{\epsilon\} \cup L(e')$, in case the empty word belongs to the language, otherwise $\Lambda = \emptyset$. It can be defined inductively for each RE as following:

$$\begin{aligned} \Lambda(\emptyset) &= \emptyset \\ \Lambda(\epsilon) &= \{\epsilon\} \\ \Lambda(a) &= \emptyset, \text{ for each letter } a \\ \Lambda(s+t) &= \Lambda(s) \cup \Lambda(t) \\ \Lambda(s\ .\ t) &= \Lambda(s)\ .\ \Lambda(t) \\ \Lambda(s^*) &= \{\epsilon\} \end{aligned}$$

– **Step 3** - computation of the local language, i.e. the set of words which begin with a letter of P, end by a letter of D and whose transitions belong to F:

$$L' = (PB^* \cap B^*D)\ \setminus\ B^*(B^2 \setminus F)B^*$$

– **Step 4** - erasing the delinearization, giving to each letter of B the letter of A.

Given the regular expression $\mathbf{e} = (\mathbf{a}(\mathbf{ab})^*)^* + (\mathbf{ba})^*$, it would produce the NDFA in Figure 3, without performing Step 4, to perform this step one only needs to remove the index from each symbol in the nodes.

Powerset construction algorithm Converts a NDFA into a DFA, which recognizes the same formal language. Recall the structure of a NDFA in Section 4.1, the corresponding DFA has the set of states Q_D, where each state corresponds to subsets of Q_N.

The initial state of the DFA is the set $\{q_0\}$ where q_0 is the initial state of the NDFA. In case of a NDFA has ϵ-transitions then the initial state of the DFA - q_{0D} - has the initial state of the NDFA - q_{0N} - plus the states reachable from that state through ϵ-transitions - called ϵ-*closure*:

$$q_{0D} = q_{0N}\ \cup\ \epsilon\text{-}closure\ \{q_{0N}\}$$

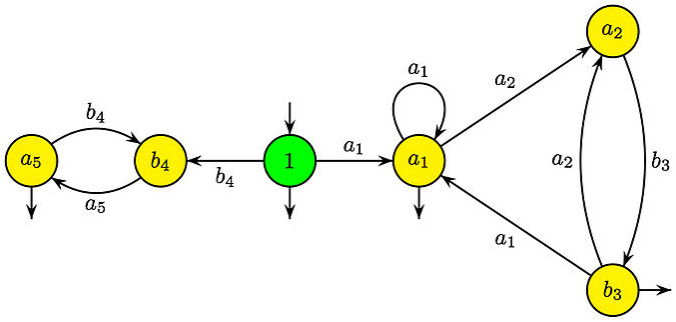


Fig. 3. NDFA resulting from $\mathbf{e} = (\mathbf{a}(\mathbf{ab})^*)^* + (\mathbf{ba})^*$ using Glushkov's construction

Starting from the initial state q_{0D}, each new state is computed from another state $S \in Q_D$. It can be defined recursively as:

- $q_{0D} \in Q_D$
- $qs \in Q_D \Rightarrow \{d \mid (o, y, d) \in \delta_{NDFA} \wedge o \in qs\} \in Q_D v, \quad \forall y \in \Sigma$

The transition function of the DFA maps a state S and the input symbol y to the respective computed state:

$$\delta_{DFA} = \cup\{(S, y, D) | D = \{d \mid (o, y, d) \in \delta_{NDFA} \wedge o \in S\}\}, \forall (y \in \Sigma \wedge S \in Q_D)$$

Finally, the set of accepting states of the DFA Z_D is the set of states $S \in Q_D$ that contain at least one state that is accepting state in the NDFA:

$$Z_D = \{q \in Q_D \mid q \cap Z_N \neq \emptyset\}$$

Kleene's Algorithm Kleene's algorithm produces a regular expression from a DFA in the following way.

1. It first constructs a graph whose nodes correspond to the states of the automaton and the edges are regular expressions that connect those nodes.
2. The initial set of edges contains one edge for each transition in the automaton (labelled with the corresponding literal) plus one edge for each node to itself containing the regular expression ϵ.
3. It proceeds by applying a variation of the Floyd-Warshall algorithm in order to obtain all paths connecting each of the nodes (summing weights corresponds to concatenation of regular expressions whereas the minimum operation corresponds to alternative (+) of regular expressions).
4. Finally the regular expression equivalent to the original DFA can be obtained as the union (+) of the edges which connect the initial state to all final states.

5 Implementation

This section describes the process of implementing BX between a RE and a DFA. It will start by describing the reasoning about how to implement the BX and will follow by explaining the implementation process, including the explanation about the algorithm's choices. Finally, it will present the *put* strategy defined for a BX between NDFA and DFA.

The source code of the implementation is available in the Github repository: `https://github.com/lisandrasilva/rv-bx`.

5.1 Reasoning about BX

Considering the already existent algorithms to convert between a RE and a DFA, we decided to attempt implementing the BX as a composition of two small BX, more concretely, one between RE and NDFA and another between NDFA and DFA. As depicted in Figure 4 the NDFA would be an intermediate structure used as a view in one of the BX and as a source in the another.

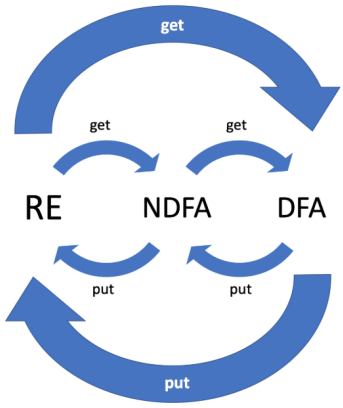


Fig. 4. BX between RE and DFA

In the BX between RE and NDFA the source is the RE and the view is the NDFA. The *getNDFA* function from RE to NDFA would be one of the conversion algorithms presented on Section 4.2 - Thompson's construction or Glushkov's construction. Regarding the BX between NDFA and the DFA, the source is the NDFA and the view is the DFA, where the *getDFA* function would be the Powerset construction algorithm.

The *get* function that given a RE as source and produces a DFA as view will be the composition of these two functions:

$$\textbf{\textit{get re}} = (\textit{getDFA} \;.\; \textit{getNDFA})\ \textit{re}$$

The main goal is to define the *putback* functions:

putNDFA - function that accepts a NDFA as a source and a possible modified DFA as a view and reflects the changes on the view into the source;
putRE - function that accepts a RE as a source and a possible modified NDFA as a view and reflects the changes on the view into the source.

The final *putback* function that accepts a RE as a source and a possible modified DFA as a view and produces the updated RE will be the composition of these two functions:

$$\textbf{\textit{put re dfa}} = \textit{putRE re (putNDFA (get re) dfa)}$$

To tackle the problem we decided to start by define the *putNDFA* function.

5.2 Get function

As said above the *get* function will be the composition of two, with a NDFA as an intermediate structure. Once this NDFA will be the source argument of the *putNDFA* function and the view argument for the *putRE* function, it is necessary to reason about what type of NDFA we should consider, which means reason about the choice for the *getNDFA* function.

The NDFA can be obtained through the Thompson's construction algorithm or through the Glushkov's construction algorithm. But, recall that the resulting NDFA are different since the one obtained through Glushkov's algorithm is ϵ-*free*, while the one obtained through the Thompson's algorithm has ϵ-transitions.

NDFA with ϵ-transitions Let's consider the NDFA in Figure 5 and its correspondent DFA in Figure 6 obtained through the Powerset construction algorithm.

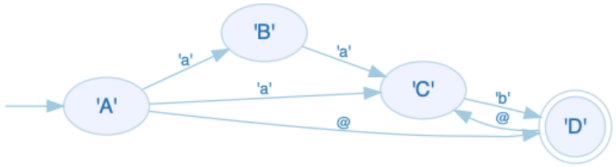


Fig. 5. NDFA with ϵ-transitions

By analyzing those figures is becomes clear that it is hard to reason about a *putback* function when the NDFA has ϵ-transitions. For instance, in the obtained

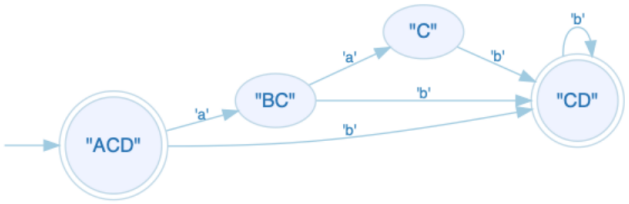


Fig. 6. DFA obtained from NDFA in Figure 5

DFA of the given example (Figure 6) all the nodes are dependent on the transitions of the 'C' node in the NDFA, which means that when adding or removing transitions from or to node 'C' in the NDFA, these changes will be reflected in all the nodes with 'C' in the DFA (because that is what is demanded by the Powerset construction).

This fact restricts too much the possible changes in the view, and even though we designed a pseudo-algorithm to *put* function in many cases it didn't satisfied the GetPut law.

NDFA *ϵ-free* Figure 7 depicts a *ϵ-free* NDFA equivalent to the NDFA in Figure 6 and in Figure 8 its the correspondent DFA.

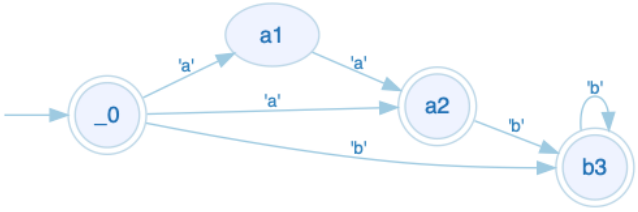


Fig. 7. NDFA *ϵ-free*

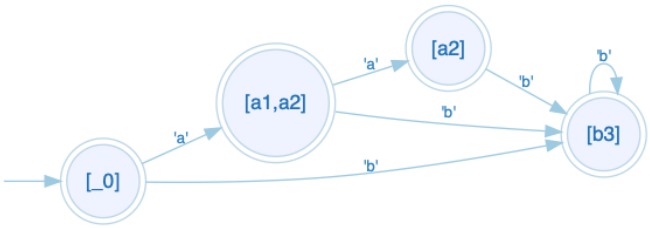


Fig. 8. DFA obtained from NDFA in Figure 5

By analyzing the two automata, we can conclude that there are less dependencies, the only dependency is between nodes `[a1,a2]` and `[a2]`, because both have the node `a2` from the NDFA. Therefore changes in this node will be reflected in both nodes in the view, even when only one was modified in the first place.

However, as it allows a wider range of possible changes on the view, we decided to choose the ϵ-*free* NDFA to define the *put* strategy between NDFA and DFA, meaning that the *get* function from RE to NDFA should be the Glushkov's construction algorithm.

5.3 Put back from DFA to NDFA

Structures for DFA and NDFA When reasoning about the put back function, the first step was to define the structures to represent both the NDFA and the DFA. Both structures are structurally very similar, but they will differ in the transition table `delta`, since in the DFA is not allowed to have two transitions from the same node with the same symbol, while in the NDFA that is possible.

```
data Ndfa st sy = Ndfa { vocabularyN :: [sy ],
                         statesN     :: [st ],
                         initialSN   :: [st ],
                         finalSN     :: [st ],
                         deltaN      :: [((st,sy),st)]
                       }

data Dfa st sy = Dfa { vocabularyD :: [sy],
                       statesD     :: [st],
                       initialSD   :: st,
                       finalSD     :: [st],
                       deltaD      :: [((st,sy),st)]
                     }
```

We decided to represent the structures with polymorphism on the arguments so they can be more flexible on the types they can represent. However, remind that in the present work, the DFA is result of the Powerset construction of the NDFA, which means that the states in the DFA are sets of the states in the NDFA. As so, if in NDFA the type of `st` is for example `Int`, then the type of `st` in the DFA is `[Int]`.

Table 1 shows how the automata in Figures 7 and 8 are (pretty-printed) represented in the data structures mentioned above.

Reflecting transition table The function `getTable` is responsible for reflecting the DFA transition table into the NDFA transition table. Therefore, it receives as argument the NDFA transition table, the DFA transition table and the set of states q of the DFA (to perform checks) and returns the updated NDFA transition table. The argument with the DFA transition table is in fact zipped

	NDFA	DFA
Vocabulary	'b', 'a'	'b', 'a'
States	_0, b3, a2, a1	[a2], [a1,a2], [b3], [_0]
Initial State	_0	[_0]
Final States	_0, a2, b3	[a2], [a1,a2], [b3], [_0]
Transition Table	$\_0 \xrightarrow{'a'} a1$ $\_0 \xrightarrow{'b'} a2$ $\_0 \xrightarrow{'b'} b3$ $a1 \xrightarrow{'b'} a2$ $a2 \xrightarrow{'b'} b3$ $b3 \xrightarrow{'b'} b3$	$[\_0] \xrightarrow{'a'} [a1,a2]$ $[\_0] \xrightarrow{'b'} [b3]$ $[a1,a2] \xrightarrow{'a'} [a2]$ $[a1,a2] \xrightarrow{'b'} [b3]$ $[a2] \xrightarrow{'b'} [b3]$ $[b3] \xrightarrow{'b'} [b3]$

Table 1. Representation of automata in Figures 7 and 8

with an accumulator that will save, for each DFA transition $os \xrightarrow{s} ds$, the subset of ds of the NDFA transitions that were validated so far.

The code presented below can be better understood with the example given afterwards.

```
getTable :: (Ord st, Ord sy) =>[((st,sy),st)]
                             ->[((([st],sy),[st]),[st)]
                             -> [[st]]
                             ->[((st,sy),st)]
getTable [] dfaT q =
    let toAdd = filter (not . null . snd)
                [(((os,d),ds\\rs)) |(((os,d),ds),rs) <- dfaT]
    in concat $ map (`rearrangeS` q)toAdd

getTable (t@((o,s),d):ts) dfaT q =
    let relS = [ x | x <- q , o `elem` x]
        trns = [ 1 | (((os,sym),ds),rs) <- dfaT ,
                                           os `elem` relS ,
                                           sym == s ,
                                           d `elem` ds]
        dfaT' = map (updateListAux t) dfaT
    in if (length relS == length trns) && (not $ null relS)
       then t:getTable ts dfaT' q
       else getTable ts dfaT q
```

For each NDFA transition `((o,s),d)`, the function verifies that each node in the DFA that contains 'o' has the transition through symbol 's' for a node that

contains '*d*'. When this is valid then the NDFA transition is kept in the source and the '*d*' is added to the auxiliary list from the DFA transitions mentioned above (using `updateListAux` function), otherwise is discarded.

```
updateListAux ((o,s1),d)(((os,s2),ds),rs) =
            if o 'elem' os && s1 == s2 &&d 'elem' ds
            then (((os,s2),ds),d:rs)
            else (((os,s2),ds),rs)
```

Going back to the example in Figures 7 and 8, let's do the following sequence of modifications on the view (depicted in DFA in Figure 9):

- Remove transition [a2] $\xrightarrow{'b'}$ [b3]
- Add transition [a1,a2] $\xrightarrow{'c'}$ [c4]
- Add transition [c4] $\xrightarrow{'b'}$ [b3]

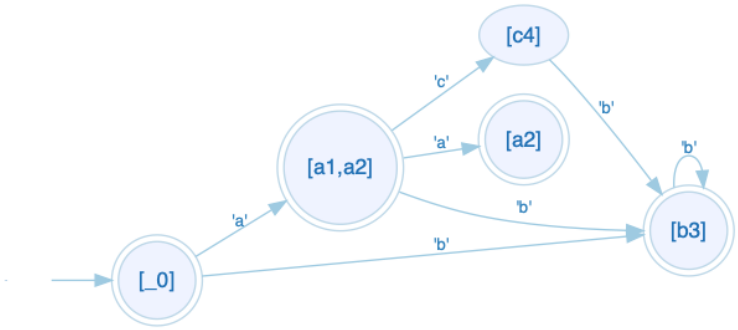


Fig. 9. Updated view

Table 2 represents the arguments to the function `getTable` with the updated DFA transition table. As mentioned, the function `getTable` goes recursively through the NDFA transitions. For the first NDFA transition `((_0,'a'),a1)` the function gets all the DFA states that contain _0 (which is only [_0]), and ensures that it has the transition through 'a' to a node which contains a1. As this is true due to the transition `(([_0],'a'),[a1,a2])` then a1 is added to the auxiliary accumulator list of the transition.

	NDFA	DFA	
States	_0, b3, a2, a1	[a2], [a1,a2], [b3], [_0], [c4]	
Transition Table	_0 $\xrightarrow{'a'}$ a1 _0 $\xrightarrow{'b'}$ a2 _0 $\xrightarrow{'b'}$ b3 a1 $\xrightarrow{'b'}$ a2 a2 $\xrightarrow{'b'}$ b3 b3 $\xrightarrow{'b'}$ b3	[_0] $\xrightarrow{'a'}$ [a1,a2] [_0] $\xrightarrow{'b'}$ [b3] [a1,a2] $\xrightarrow{'a'}$ [a2] [a1,a2] $\xrightarrow{'b'}$ [b3] [a2] $\xrightarrow{'b'}$ [b3] [b3] $\xrightarrow{'b'}$ [b3] [a1,a2] $\xrightarrow{'c'}$ [c4] [c4] $\xrightarrow{'b'}$ [b3]	[ ] [ ] [ ] [ ] [ ] [ ] [ ]

Table 2. Arguments for `getTable` with updated view

When analyzing NDFA transition `((a2,'b'),b3)`, all the DFA nodes that contain `a2` (`[a1,a2]` and `[a2]`) must have the transition through `'b'` to a node that contains `b3`. However this is not verified since the DFA transition `(([a2],'b'),[b3])` was removed from the view, therefore this NDFA transition is not kept in the source.

	NDFA	DFA	
States	_0, b3, a2, a1	[a2], [a1,a2], [b3], [_0], [c4]	
Transition Table	_0 $\xrightarrow{'a'}$ a1 _0 $\xrightarrow{'b'}$ a2 _0 $\xrightarrow{'b'}$ b3 a1 $\xrightarrow{'b'}$ a2 a2 $\xrightarrow{'b'}$ b3 b3 $\xrightarrow{'b'}$ b3	[_0] $\xrightarrow{'a'}$ [a1,a2] [_0] $\xrightarrow{'b'}$ [b3] [a1,a2] $\xrightarrow{'a'}$ [a2] [a1,a2] $\xrightarrow{'b'}$ [b3] [a2] $\xrightarrow{'b'}$ [b3] [b3] $\xrightarrow{'b'}$ [b3] [a1,a2] $\xrightarrow{'c'}$ [c4] [c4] $\xrightarrow{'b'}$ [b3]	[a1,a2] [b3] [a2] [ ] [b3] [ ] [ ]

Table 3. Transition tables after `getTable` processed all old NDFA transitions

Table 3 contains the state of the transition tables after the function `getTable` already processed all the NDFA transitions. After that, the function `getTable` is recursively called with the first argument (NDFA table transtion) empty. At this point, for each DFA transition `((os,s),ds)` its corresponding auxiliary list with the subset of *ds* that were already validated is removed from the set *ds*.

If the remaining *ds* list is empty then it means that the transition was already validated. When there are still states in the list *ds*, the new transitions in the NDFA must be created (Table 4). This is done in the `rearrangeS` function.

	NDFA	DFA
States	_0, b3, a2, a1	[a2], [a1,a2], [b3], [_0], [c4]
Transition Table	_0 $\xrightarrow{'a'}$ a1 _0 $\xrightarrow{'b'}$ a2 _0 $\xrightarrow{'b'}$ b3 a1 $\xrightarrow{'b'}$ a2 b3 $\xrightarrow{'b'}$ b3	[a1,a2] $\xrightarrow{'b'}$ [b3] [ ] [a1,a2] $\xrightarrow{'c'}$ [c4] [ ] [c4] $\xrightarrow{'b'}$ [b3] [ ]

Table 4. Transition tables before call `rearrangeS` function

The function receives a DFA transition, the set of DFA states and creates the list of new NDFA transitions corresponding to the given DFA transition. First the function computes the minimum list of NDFA transitions, excluding transitions whose origin appears in other DFA nodes.

```
rearrangeS :: Eq st => (([st], sy), [st])
                    -> [[st]]
                    -> [((st, sy), st)]
rearrangeS ((os,sy),dsts) q =
    let min = [((o,sy),dd) | o <- os \\ (concat $ delete os q),
                             dd <- dsts]
    in if null min then [((o,sy),dd) | o <- os, dd <- dsts]
       else min
```

For instance, when creating the new NDFA transitions for the DFA transition `(([a1,a2],'b'),[b3])` it is only desirable to create the new transition `((a1,'b'),b3)` since, if a transition from `a2` is also created then, when *getting* again the DFA, the node `[a2]` will also have that transition, which was not created by the user, leading to a violation of the GetPut law.

Sometimes it is not possible to create new transitions without violating the law. For instance, if instead of the DFA transition `(([a1,a2],'b'),[b3])` we had `(([a2],'b'),[b3])`, the minimum computed list of the new NDFA transitions would be empty. This means that the edited view is not consistent with the source, because it is not possible to reflect the modifications without modifying other node transitions. In these cases, the function returns all the possible transitions and the inconsistency will be handled later.

	NDFA
Transition Table	$\_0 \xrightarrow{'a'}$ a1 $\_0 \xrightarrow{'b'}$ a2 $\_0 \xrightarrow{'b'}$ b3 a1 $\xrightarrow{'b'}$ a2 b3 $\xrightarrow{'b'}$ b3 a1 $\xrightarrow{'b'}$ b3 a1 $\xrightarrow{'c'}$ c4 c4 $\xrightarrow{'b'}$ b3

Table 5. Updated NDFA transition table

Updating NDFA Structure After the transition table of the NDFA is complete, it is still necessary to update the other fields of the NDFA structure. This is done by the function `putNdfaStruct`.

The *put* of the other fields of the structure is very straightforward, the most complex being the final states. For that, for each *old* final state, the function tests if it still belongs to some of the new final states, and in that case, it is kept as a final state or discarded otherwise. In the end the remaining final states are added as final states.

```
putNdfaStruct :: (Ord st, Ord sy) => Ndfa st sy
                                  -> Dfa [st] sy
                                  -> Ndfa st sy
putNdfaStruct (Ndfa v1 q1 s1 z1 d1) (Dfa v2 q2 s2 z2 d2) =
        Ndfa v q s z d
    where v = v2
          q = nub $ concat q2
          s = s1
          z = f z1 (concat z2)
          d = getTable d1 (zip d2 (repeat [])) q2
          f [] cz2 = cz2
          f (h:t) cz2 = if h `elem` cz2
                        then h:(f t (filter (/= h) cz2))
                        else f t cz2
```

Table 6 shows the updated NDFA (source) given the modified DFA (view), which is the result of the function `putNdfaStruct`.

	NDFA	DFA
Vocabulary	'b', 'a', 'c'	'b', 'a', 'c'
States	_0, b3, a2, a1, c4	[a2], [a1,a2], [b3], [_0], [c4]
Initial State	_0	[_0]
Final States	_0, a2, b3	[a2], [a1,a2], [b3], [_0]
Transition Table	_0 $\xrightarrow{'a'}$ a1 _0 $\xrightarrow{'b'}$ a2 _0 $\xrightarrow{'b'}$ b3 a1 $\xrightarrow{'b'}$ a2 b3 $\xrightarrow{'b'}$ b3 a1 $\xrightarrow{'b'}$ b3 a1 $\xrightarrow{'c'}$ c4 c4 $\xrightarrow{'b'}$ b3	[_0] $\xrightarrow{'a'}$ [a1,a2] [_0] $\xrightarrow{'b'}$ [b3] [a1,a2] $\xrightarrow{'a'}$ [a2] [a1,a2] $\xrightarrow{'b'}$ [b3] [b3] $\xrightarrow{'b'}$ [b3] [a1,a2] $\xrightarrow{'c'}$ [c4] [c4] $\xrightarrow{'b'}$ [b3]

Table 6. Representation of automata in Figures 7 and 8

Well-built DFA As the DFA can be edited by the user, and as these changes will be reflected in the NDFA, it is important to guarantee that the DFA is well built before proceeding with the *put* function. This is guaranteed by the function `wellBuilt` that ensures the following:

- Every transition from a given origin o with a symbol s is unique;
- Every Node x must be reachable from the initial node;
- Every node must reach an accepting node;
- For every transition $o \xrightarrow{s} d$, each state in the destination's list of d states must be prefixed by the symbol s (considering the NDFA is result of Glushkov's algorithm and DFA is result of Powerset construction);
- The initial state cannot be modified (also a requirement of Glushkov's algorithm)

The function returns an error message in case any of the above mentioned rules is violated.

Put NDFA Finally, the function `putNDFA` puts it all together: it receives the NDFA, the (possibly) updated DFA and returns an `Error NDFA`. The `Error NDFA` is a data type that can contain the (updated) NDFA if everything went well or an error message in case the given DFA was not a valid DFA. The verification of the validity of DFA is done by the function `wellBuilt` that returns an `Error Bool`, i.e. an `Ok True` case the DFA is valid or an `Error m` where `m` contains the error message.

```
putNdfa :: Ndfa (Indexed Char) Char
        -> Dfa [Indexed Char] Char
        -> Error (Ndfa (Indexed Char) Char)
putNdfa ndfa dfa = case wellBuilt dfa of
                     Ok True -> Ok (putNdfaStruct ndfa dfa)
                     Error m -> Error m
```

Given this `put` algorithm if we call the function `putNdfa` with the NDFA and DFA represented in Table 1 (unmodified view), the output will be the same NDFA (Figure 7), which means that the PutGet law is satisfied. If we call the function `putNdfa` with the NDFA and DFA represented in Table 2 (updated view), and then call the *get* function (powerset construction) the DFA produced will be the same (Figure 9), which means that for the given example the GetPut law is satisfied. However, as mentioned before this doesn't happen when there is inconsistencies between source and view, next section explains how the program deals with them.

Inconsistencies Taking again the previous example, it was possible to remove the transition `(([a2],b),[b3])` without violating the GetPut law property. However if we had removed the transition `(([a1,a2],b),[b3])` this would result in an inconsistency, because if the transition `(([a2],b),[b3])` still exists, it means that the NDFA will have the transition `((a2,b),b3)` and by the Powerset construction algorithm the transition `(([a1,a2],b),[b3])` will still be there.

When an inconsistency occurs in the view, it means that is not possible to get the same view with the updated source, because the dependencies in the view were not 'respected'. In these cases, the user is asked if he wants to proceed with the consistent version of the view, and in that case the source is updated with the consistent version of the view.

Interpreter It was also created an interpreter to interact with the user. The interpreter allows the user to perform a sequence of modifications on the view, that are displayed in graphs so the user can see the updates being done. The possible modifications that the user can perform are:

- Add transition;
- Remove transition;
- Remove node;

At any point the user can call the *put*, even when he had not performed any modification. In case the sequence of modifications results in an updated view that is not consistent with the source, the interpreter asks the user if he wants to continue or to rollback. In case he wants to continue the source is updated with the consistent version of the view, otherwise the modifications are discarded.

6 Conclusions and Future work

The initial goal was to implement the BX as a composition of two BX's, one between RE and NDFA and another between NDFA and DFA. However, only the second one was defined. As future work it is still necessary to implement the first one, between RE and NDFA, considering as the *get* function the Glushkov's construction algorithm.

Another possibility worth exploring would be to use other forms of transforming RE into finite state machines, namely those based in Brzozowski derivative [3]. These transformations generate a DFA whose states are themselves RE that capture the language to be recognized starting at that state. This holographic feeling of the produced automata make them particularly suited to think in the transformations as BX.

As said in Section 3, this was an exploratory work that used the 'naive' way to implement a BX where the two functions *get* and *put* separately. To avoid the need of a formal proof that the two functions are *well-behaved* it would be very useful to use a *bidirectional programming* language to implement the put algorithm. We suggest a *putback based* approach, more concretely BiGUL.

Finally, some work has being done in translation between specification languages, a challenge for the future could be implement synchronizations between another specification languages, namely Linear Temporal Logic and Context Free Grammars.

Bibliography

[1] Bartocci, E., Falcone, Y., Francalanza, A., and Reger, G. (2018). Introduction to runtime verification.

[2] Bohannon, A., Foster, J. N., Pierce, B. C., Pilkiewicz, A., and Schmitt, A. (2007). Boomerang: Resourceful lenses for string data. Technical report, Department of Computer and Information Science, University of Pennsylvania.

[3] Brzozowski, J. A. (1964). Derivatives of regular expressions. *JOURNAL OF THE ACM*, 11:481–494.

[4] Falcone, Y., Fernandez, J.-C., and Mounier, L. (2011). What can you verify and enforce at runtime verification?

[5] Falcone, Y., Havelund, K., and Reger, G. (2012). A tutorial on runtime verification.

[6] Gruber, H. and Holzer, M. (2014). From finite automata to regular expressions and back - a summary on descriptional complexity.

[7] Hu, Z. and Ko, H.-S. (2014). Principles and practice of bidirectional programming in bigul. Technical report, National Institute of Informatics.

[8] Zan, T., Liu, L., Ko, H.-S., and Hu, Z. (2016). Brul: A putback-based bidirectional transformation library for upadate views. In *Proceedings of the Fifth International Workshop on Bidirectional Transformations*.