A proof-of-concept prototype for IFTA
Technical Report

Guillermina Cledou

HASLab INESCTEC and Universidade do Minho, Braga, Portugal

mgc@inesctec.pt

Abstract

1 Introduction

This report presents a proof-of-concept prototype implemented to support Interface Featured Timed Automata (IFTA) [3], a compositional formalism to model Software Product Lines (SPLs) with time requirements, and discusses some implementation decisions and challenges encounter when experimenting with the prototype. The relevant background on SPL and IFTA can be found in [1–3].

The prototype is developed in Scala[1] and consists of a small Domain Specific Language (DSL) to support specification of IFTA and networks of IFTA (NIFTA), as well as some operations over them. By a network of any kind of automata we understand a set of automata composed in parallel ($||$) and synchronized over a set of shared actions.

2 Prototype

The DSL provides an easy way to specify IFTA and networks of IFTA, and provides different features to work over both, including:

- *operations:* product, synchronization and composition
- *translations to other formalisms:* (networks of) FTA and Uppaal (networks of) TA
- *graphical representations:* statical – using DOT[2] graph description language; and interactive – using Vis.js[3] visualization library
- *connectors:* a set of commonly used $\mathcal{R}$eo connectors to coordinate *families* of systems

We informally explain the DSL and its use through a simple example.

2.1 Example: family of payment methods

Let us consider a family of systems that supports online payments through different methods. Some services support online payment through *credit cards* (cc), other services support payments through *PayPal* (pp), and others support both. This represents a family of three possible services with shared functionality, namely $\{cc, pp\}, \{cc\}$, and $\{pp\}$.

We model this small family by separating the behaviour of each type of payment into an automaton. The mechanism that coordinates the selection of the payment method is also separated into different automata. The behaviour of each of these components is described below.

[1] https://github.com/haslab/ifta
[2] http://www.graphviz.org/about/
[3] http://visjs.org

Credit Card – the component[4] models payments through credit cards. If a user requests to pay by credit card (*paycc*), a clock is reset to track payment elapsed time (*topp*), after which the user has less than 1 unit of time ($Inv_{CC}(\ell_1)$) to enter the details and proceed with the payment, which can result in success (*paidcc*) or cancellation (*cancelcc*).

PayPal – the component models payments through PayPal accounts. If a user requests to pay by PayPal (*paypp*), a clock is reset to track payment elapsed time (*tocc*), after which the user has less than 1 unit of time ($Inv_{PP}(\ell_1)$) to login and proceed with the payment, which can result in success (*paidpp*) or cancellation (*cancelpp*).

PaymentNet – It comprises a network of automata to model the orchestration of payment requests based on the availability of payment methods. It is composed by the Credit Card and PayPal components, a *router* and two *merger* connectors. If a request for payment is received (*payapp*) a *router* enables to choose between a payment by Credit Card or by PayPal (*paypp* or *paycc*). A *merger* synchronizes the successful response (*paidpp* or *paidcc*), while other *merger* synchronizes the cancellation response (*cancelpp* or *cancelcc*) from either *CC* or *PP*. In addition to the feature model generated by the composition of these IFTA, the payment component imposes the additional restriction that at least one payment method is supported by the system, ($cc \vee pp$).

A scheme of the payment network can be seen in Figure 1. This figure was generated by the prototype as we will explain below. Boxes represent IFTA, while ellipses represent the interface of the network, i.e. input and output ports that haven't been synchronized yet. In addition, a graphical representation of each IFTA, automatically generated by the prototype, can be seen in Figure 4.

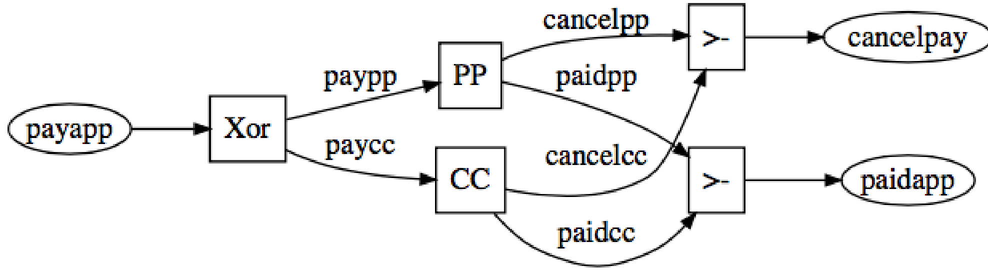


Figure 1: A architectural view of a network of IFTA.

2.2 Specifying IFTA

This network can be specified in the prototype by using the DSL as shown in Listing 1. In the figure, `creditcard`, `paypal`, and `paymentNet`, correspond to the components described above. A new automaton is created with the constructor `newifta`, which builds an empty IFTA. New transitions are added through the operator `++`. In the case of the credit card IFTA, three transitions are specified between parenthesis, followed by the declaration of the ports. Each transition in the example specifies:

- the origin and destination location indicated by natural numbers and by the linking operator between them `->`;
- the actions labelling the transition, using the operator `by` followed by the set of actions encoded as a string where actions are separated by a comma;
- the associated feature expression, using the operator `when`, followed by the feature expression, where features are expressed as strings and the logical operators are encoded as usual, `&&`, `||`, `->`, `not`, and `<->`; and
- the set of clock to reset, using the operator `reset`, preceding each clock to reset, encoded as a string, and any clock constraint if it were the case, e.g., `cc "t">=2`, where `cc` stands for *clock constraint*, and `t` is a clock name.

[4]the word component is used interchangeably with automaton.

```
val creditcard = newifta ++ (
    0 --> 1 by "paycc" when "cc" reset "toutcc",
    1 --> 0 by "cancelcc" when "cc",
    1 --> 0 by "paidcc" when "cc"
    ) startWith 0 get "paycc" pub "cancelcc,paidcc" inv(1,"toutcc"<=1) name "CC"

val paypal = newifta ++ (
    0 --> 1 by "paypp" when "pp" reset "toutpp",
    1 --> 0 by "cancelpp" when "pp",
    1 --> 0 by "paidpp" when "pp"
    ) startWith 0 get "paypp" pub "cancelpp,paidpp" inv(1,"toutpp"<=1) name "PP"

val paymentNet = (
    router("payapp", "paycc", "paypp") ||
    paypal || creditcard ||
    merger("cancelcc", "cancelpp", "cancelpay") ||
    merger("paidcc", "paidpp", "paidapp")) when ("pp" || "cc")
```

Listing 1: Example specification of a network of IFTA using the prototype DSL.

Each port, encoded as a string, is preceded by the operator `get` if it is an input port, and `pub` if an output port. The feature model is specified by a feature expression using the operator `when` as before. When the feature model is not specified, it is automatically assumed as $\top$. Similarly, if the initial location is not specified by `starWith`, location `0` is assumed as initial location. A net can be created by composing automata in parallel as in the case of `paymentNet`. The DSL provides a set of $\mathcal{R}$eo connector constructs, such as the `router` and `merger` used in the example. In this case, both connectors follow the *relaxed* approach [1] in which some of their ports may be missing.

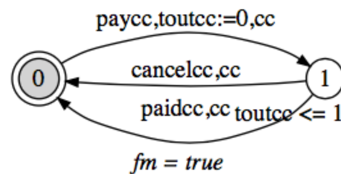
2.3 Graphical representation

The prototype provides functionality to visualize IFTA and networks of IFTA, either statically or interactively. In the statical option, automata are translated to *Dot*, a graph representation language. In the interactive option, automata are as well translated to a graph representation language but using a JavaScript library, *Vis.js*. For example, the command `toDot(creditcard)` produces the following output code in Dot language (left) which can be visualized as the graph on the right[5]:

```
digraph G {
  rankdir=LR;
  node [margin=0 width=0.3 height=0.2]
  edge [arrowsize=0.7]
{ rank=min;
  node [style=filled,shape=doublecircle] 0 }
  label=<<I>fm = true</I>>

{ node [xlabel="toutcc <= 1"] 1 }

0 -> 1 [label="paycc,toutcc:=0,cc"]
1 -> 0 [label="cancelcc,cc"]
1 -> 0 [label="paidcc,cc"]
}
```

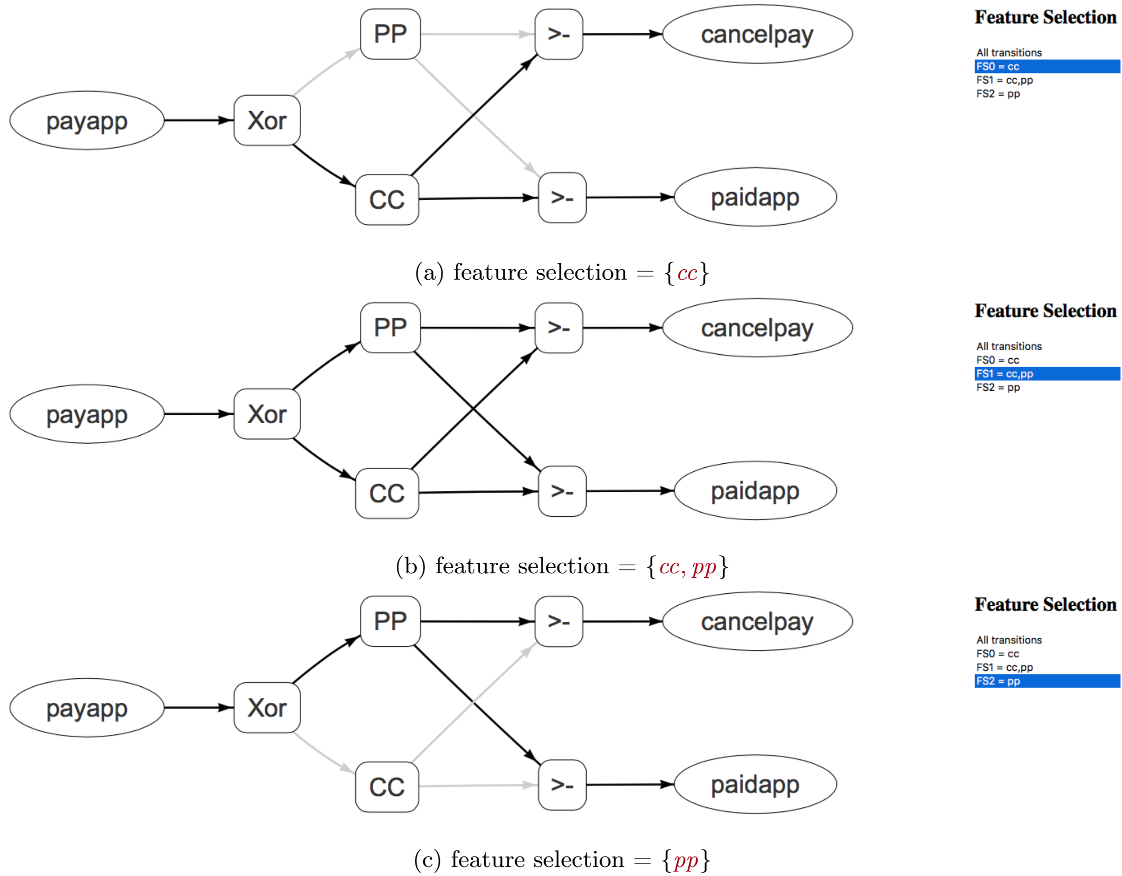


Similarly, the command `con2dot(paymentNet)` can be used over a network of IFTA. In this case, when visualized, the output code results in the graph from Figure 1.

Likewise, the commands `toVis(creditcard,"cc.html")` and `con2vis(paymentNet,"pn.html")` can be used over IFTA and networks of IFTA, respectively, to generate an HTML file. The file can be opened on any browser to visualize the corresponding IFTA or NIFTA, the set of products that can be derived from them, and how each valid feature selection affects the automaton transitions or network

[5]The code must be run with the command `dot` in the command line, or used in online tools such as http://viz-js.com/

connections. For example, in the case of the payment network, the prototype resolves there are three possible feature selections (Figure 3). By selecting each option, the model highlights in black the port connections between IFTA in the network, while grey lines represent connections that are not part of such feature selection.



(a) feature selection = $\{cc\}$

(b) feature selection = $\{cc, pp\}$

(c) feature selection = $\{pp\}$

Figure 3: Interactive visualization of a network of IFTA.

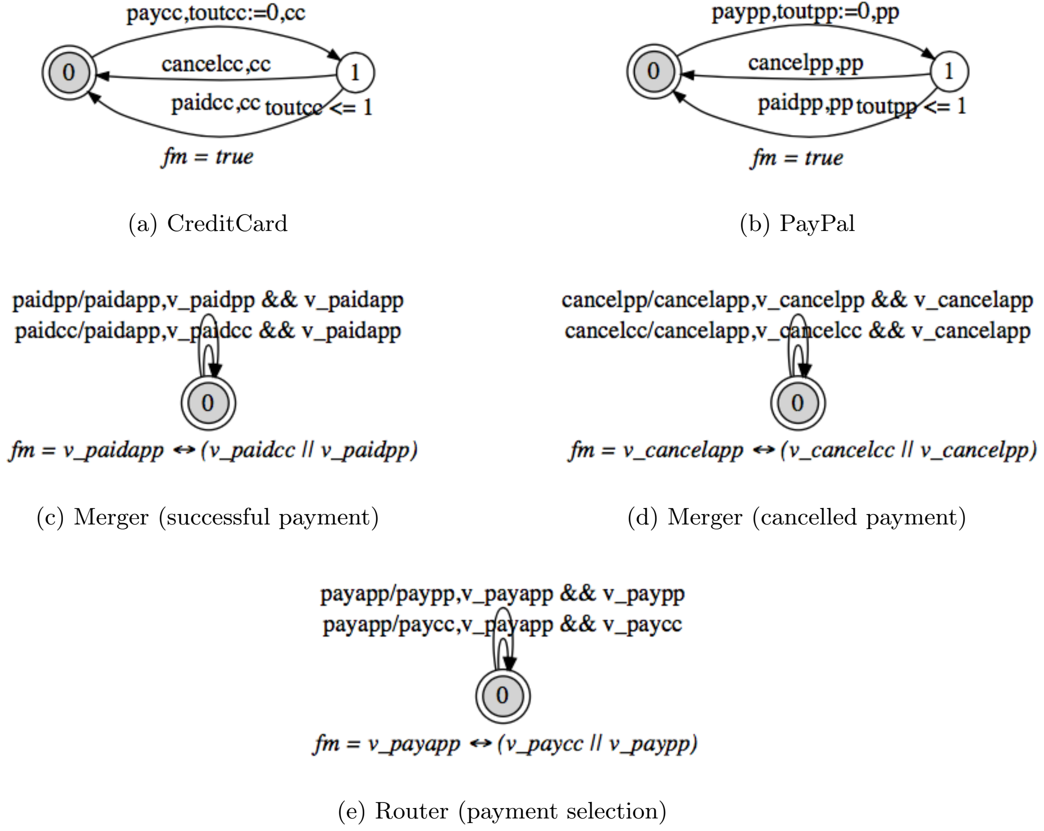
The interactive visualization is essential when modelling complex families of systems. It quickly provides visual feedback to understand if the resulting SPL models the set of desired products, and if each product has the expected underlying architecture – when visualizing networks of IFTA; or if it has the expected transitions – when visualizing IFTA.

This becomes more evident when modelling families of complex coordination mechanism due to the need of using different approaches with different degrees of variability to model each connector. As discussed in [1], currently this involves a manual process where the modeller must decide depending on the expected variability which degree of variability must exhibit each individual connector. Being able to quickly visualize how the chosen variability of a given connector affects the composed connector facilitates the modelling of such mechanisms.

2.4 Conversion to other formalisms

A network of IFTA can be step-wisely converted into a network of FTA with committed states, which in turn can be converted into a network of Uppaal TA, as follows.

1) NIFTA to NFTA. Informally, this is achieved by converting each multi-action transition to a set of transitions with single actions that must execute atomically. The atomicity is achieved through committed states between them. In addition, the new set of transitions should support all possible combinations of execution order in the original multi-action transition. However, in practice this can quickly lead to a state explosion. To reduce this problem, we allow only combinations were the execution flows from inputs to outputs. For example, in Figure 5, the IFTA on the left is converted into the FTA



(a) CreditCard

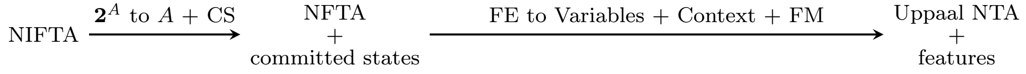
(b) PayPal

(c) Merger (successful payment)

(d) Merger (cancelled payment)

(e) Router (payment selection)

Figure 4: IFTA models for each automaton in the payment network.

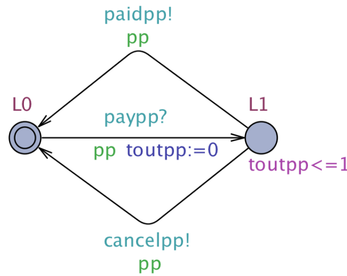


on the right. Ideally, i and o should be enabled at ℓ_0, however, we only model the combination were i executes followed by o (dark arrows). The other possibility, when o executes first (light grey) is not created. It is worth to mention that the word *first* refers here to the syntactic order, since semantically, due to the committed locations, the sequence of transitions is done atomically, i.e. time does not pass.

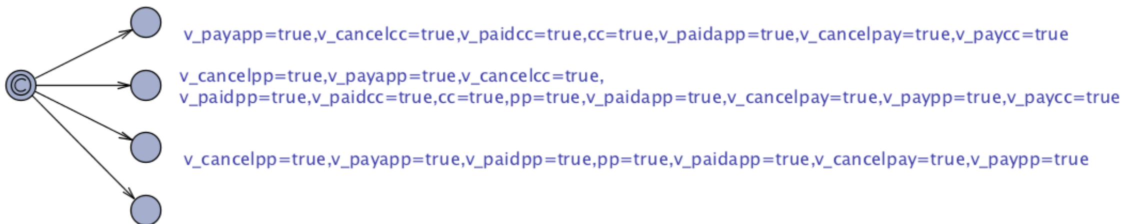
Figure 5: Conversion of multi-action transitions into single-action transitions.

2) NFTA to Uppaal NTA. To create a network ready for simulation and verification in Uppaal, this activity involves three steps. Firstly, takes the NFTA obtained in the previous step and creates an Uppaal TA per each FTA in the network, where features are encoded as boolean variables, and transition feature expressions as a logic guard over Boolean variables. Secondly, the *feature model* of the network is solved using a SAT solver to find the set of all valid feature selections (or products). This set is encoded as a TA with an initial committed location and with an outgoing transition to a new location for each element in the set. Each transition represents a valid selection of features by initializing the corresponding set of variable representing those features. The initial committed state of the feature model ensures a feature selection is made before any other transition is taken. Thirdly, a TA is created to represent the

context of the network, which corresponds to the interface of the network. The context is represented as TA with an unique state and a loop transition for each action of the context. Figure 6 illustrates how the IFTA of the PayPal component, and the feature model of the payment net (Listing 1) are translated into UPPAAL as TA. The composed feature model allows four products, from top to bottom: 1) feature *cc*, 2) features *pp* and *cc*, 3) feature *pp*, and 4) none. The additional features modelled with a prefix v_, represent generic features associated to the connectors.



(a) PayPal as UPPAAL TA



(b) Feature model as UPPAAL TA

Figure 6: Example of an Uppaal TA consisting of the *PayPal* component

A main issue when translating networks of IFTA to networks of UPPAAL TA are sequences of committed states. Because of how UPPAAL deals with such sequences, it can lead to a deadlock when verifying properties. When the model checker *sees* that the first transition in such type of sequence is enabled, it executes the transition, moving to the next committed state. If there are no enabled transitions from that state, or another committed state, then the system is in a deadlock. This is a problem inherent to UPPAAL rather than a problem of the model.

Another issues when translating IFTA to FTA with committed states, is that the complexity of the model grows quickly. For example, the IFTA of a simple replicator with 3 output ports consists of a location and seven transitions, while its corresponding FTA consists of 23 locations and 38 transitions. Without any support for composing variable connectors, modelling all possible cases is error prone and it quickly becomes unmanageable.

This simplicity in design achieved through multi-action transitions leads to a more efficient approach to translate IFTA to UPPAAL TA in particular by using the composition of IFTA. The IFTA resulting from composing a network of IFTA, can be simply converted to an FTA by *flattening* the set of actions in to a single action, and later into an UPPAAL TA, avoiding the use of committed states. For example, the payment network can be composed and flattened into a single IFTA as shown in Figure 9.

The *flattening* of the net resolves the problem of the sequence of committed states. However, it introduces a new issue, computing the composition of complex networks. For example, the prototype is not able to compute the composition of a *synchronous merger* [5, 6]. The architectural view of such coordination mechanism can be seen in Figure 7.

In the particular case of coordination mechanism, this issue is inherent to the composition of $\mathcal{R}$eo connectors and has been explored before [4]. The composition of $\mathcal{R}$eo connectors is an example where the composition of automata has a number of states/transitions linear in the number n of automata being composed, however it may required exponential resources to compute such composition. Intuitively, as connectors are being composed in Figure 7, initially the number of states and transitions may growth exponentially because these connector share only some actions or not actions at all (in the worst case it generates the product automata between two automaton). However, eventually, as all automata are

composed and began to share actions, the complexity of the composed automata reduces, since transitions begin to be synchronized due to the shared actions.

In [4] the authors exemplify this issue by composing the *constraint automata* of a `fifofull`, two `fifo1` and a `sync` connector, as depicted in Figure 8. The figure shows how the connectors are being composed (top) and the resulting automata of the composition (bottom). The results are analogous when composing the corresponding IFTA of such $\mathcal{R}$eo connectors. As future work, we intend to explore the use of a state-by-state composition approach as suggested in [4] to improve composition times.

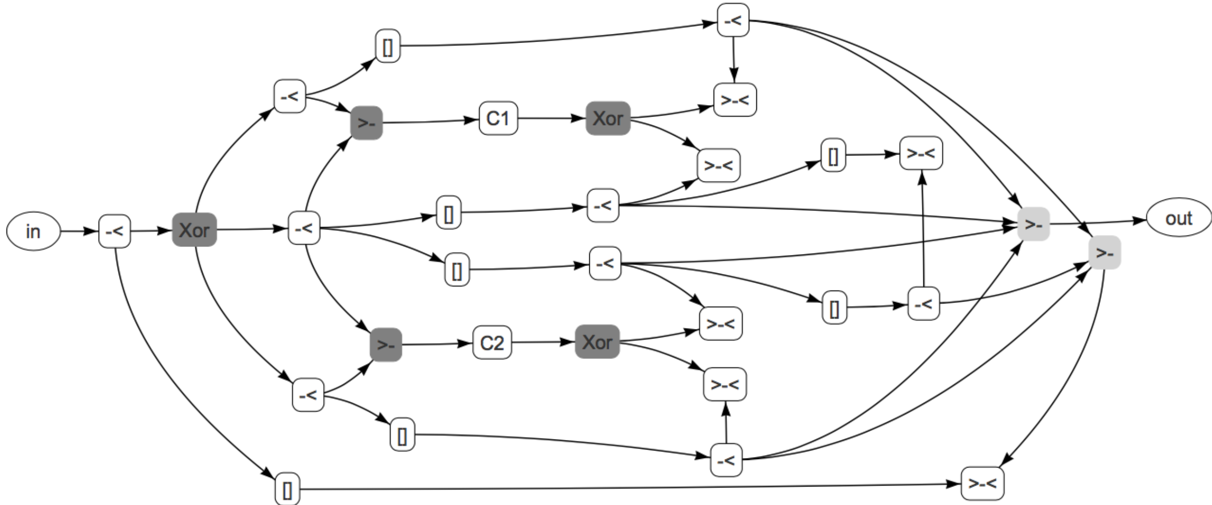


Figure 7: Synchronous merger with support for variable components.

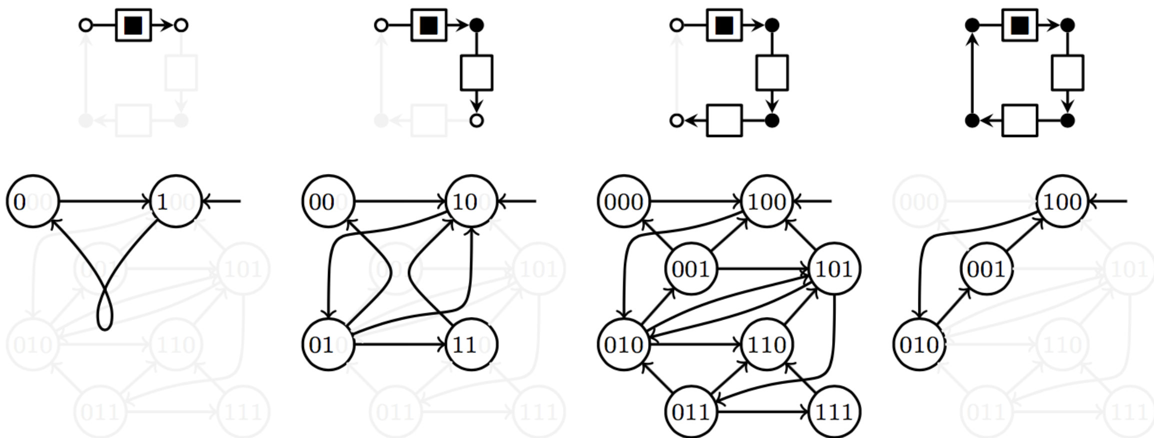


Figure 8: Example of state explosion and eventual simplification in composing $\mathcal{R}$eo connectors [4].

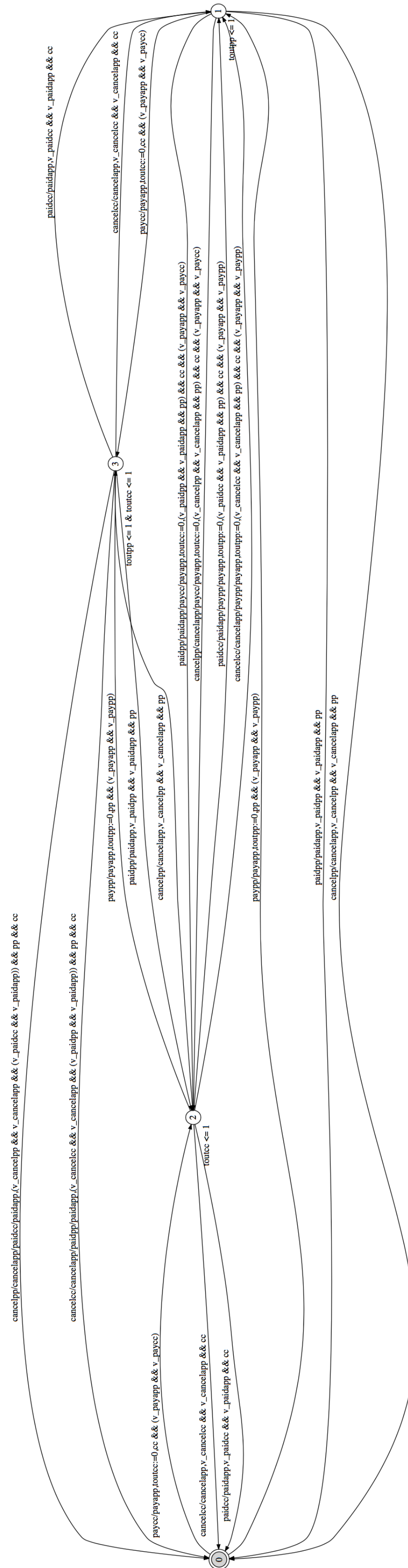


Figure 9: IFTA resulting from computing the composition of the payment network in Figure 1.

References

[1] G. Cledou. *A Virtual Factory for Smart City Service Integration (forthcoming)*. PhD thesis, Universidades do Minho, Aveiro and Porto (Joint MAP-i Doctoral Programme), 2018.

[2] G. Cledou, J. Proença, and L. S. Barbosa. A refinement relation for families of timed automata. In S. Cavalheiro and J. Fiadeiro, editors, *Formal Methods: Foundations and Applications*, pages 161–178, Cham, 2017. Springer International Publishing.

[3] G. Cledou, J. Proença, and L. Soares Barbosa. Composing families of timed automata. In M. Dastani and M. Sirjani, editors, *Fundamentals of Software Engineering*, pages 51–66, Cham, 2017. Springer International Publishing.

[4] S.-S. T. Jongmans, T. Kappé, and F. Arbab. Composing constraint automata, state-by-state. In *Revised Selected Papers of the 12th International Conference on Formal Aspects of Component Software - Volume 9539*, FACS 2015, pages 217–236, Berlin, Heidelberg, 2016. Springer-Verlag.

[5] J. Proença. *Synchronous Coordination of Distributed Components*. PhD thesis, 2011.

[6] W. M. van Der Aalst, A. H. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and parallel databases*, 14(1):5–51, 2003.