Modeling Families of Public Licensing Services: A Case Study

Guillermina Cledou
HASLab INESCTEC and Universidade do Minho
mgc@inesctec.pt

Luis Soares Barbosa
UNU-EGOV, United Nations University and Universidade do Minho
barbosa@unu.edu

***Abstract*—Software Product Lines (SPLs) enable the development of families of software systems by taking advantage of the commonalities and variabilities of the members of the family. Despite its many advantages, it is an unexplored area in the electronic government domain, an area with evident families of services, and with high demands to develop faster and better services to citizens and businesses while reducing costs. This paper discusses the need of formal methods to model SPLs for such domain. It presents a case study of a family of public licensing services modeled in UPPAAL and based on Featured Timed Automata, an extension of Timed Automata to model real-time SPLs. It analyzes the suitability of FTA to model distributed families of services, while provides hints on a possible enrichment of FTA to better support modularization and compositionality of services.**

I. Introduction

SPLs enable the definition of families of systems where all members share a high percentage of common features while differ in others. Benefits of adopting a SPL approach include reduced time to market and development costs, increased quality, and mass customization.

Although not yet widely used for modeling electronic government (e-government) services, the concept of SPL offers unique opportunities for the rapid development of certain services provided by local governments. For example, in most public administration systems, local governments are responsible for issuing licenses for public transport services. While such services comprise a number of common functionality — e.g. submission and approval of the public transport routes, vehicles' certificates, driving licenses, and even business processes that are shared with many other licensing services outside the transport domain; they also differ in a number of features, mostly due to specific regulations imposed by each local government. Therefore, the idea of applying a SPL approach for generating families of certain type of public services is appealing and relevant to the public sector.

Among several formalisms developed to support SPLs, Featured Timed Automata (FTA) captures the variability of features influencing timed and discrete behaviour based on the structure of a timed automata [1]. Relying on the concept of extending timed automata with variability, this approach enables the verification of the entire SPL instead of requiring a product-by-product verification process.

This paper presents a case study in modeling a family of licensing services for the transport domain resorting to the FTA formalism and UPPAAL[1], a well known real-time model checker. We present an approach to verify temporal properties of FTA with UPPAAL and provide examples of verifiable properties for the case study. Finally, the paper explores advantages and limitations of modeling these kind of services with FTA, and discusses possible extensions to the formalism to better support a compositional and modular specification of families of services. In particular, when modeling families of distributed services, it is necessary to model mechanisms to orchestrate variable services that adapt correctly to the presence or absence of such services. We illustrate how this task can quickly become cumbersome and error prone without further support. Furthermore, many of these orchestration mechanisms are well known coordination patterns and can be reused in different application domains. Thus, we hint on a possible extension to model generic variable orchestration mechanisms that automatically adapt to the variability of the orchestrated services through composition.

The structure of this document is as follows. Section II presents some background and motivation on electronic government, and Section III some technical background. Section IV introduces the case study, and presents the resulting models. Section V discusses the advantages and limitations of FTA and hints on possible extensions to enrich it. Section VI presents related work. Finally, Section VII concludes.

II. Modeling for E-Goverment

Electronic, or digital, government deals with the use of Information and Communication Technologies (ICTs) to facilitate the delivery of electronic public services (EPSs) and support the interaction between their providers and consumers. Among the main motivations to implement EPSs is the reduction of costs and administrative burdens [2].

However, in order to achieve these goals, governments face several challenges, namely: 1) *rapid development of EPSs* – to attend increasing citizens' demands and quickly integrate changes in regulations, government must find mechanisms to rapidly develop EPSs; 2) *service integration* – government agencies, and other entities, must collaborate to deliver *seamless* services, i.e., services delivered collaborative by several government and non-government organizations while presenting a single-organization interface to customers. The *only-once*

[1] http://www.uppaal.org

principle, which means that citizens, businesses, and other stakeholders, are required to provide common information only once, is particularly critical; 3) *conformance with laws and regulations* – the delivery of services generally depends on laws and regulations. Government must have mechanisms to ensure that an EPS conforms with such laws and regulations, otherwise, failing to correctly design and implement EPSs can increase bureaucracy, result in unused services or malfunctioning of services; and 4) *development costs* – the adoption of ICT for development of EPSs involves high costs for governments, particularly at the city level, at short term.

In practice, many of these services still relay on paper-based solutions, particularly in less resourceful governments. In many other cases, due to the differences in government regulations, lack of interoperability, budgetary resources, and the difficulties in ensuring the fulfillment of their specific features, local governments develop tailored ICT solutions to automate the provision of these services. This approach exacerbates the aforementioned challenges by increasing development times and costs, as well as hindering service integration. In addition, it disregards the fact that many public services share common functionality and business processes.

Thus, formal methods and SPLs can help to overcome, to some extent, the aforementioned challenges. On the one hand, formal methods help to model and verify that services conform with the required laws and regulations at an early stage. On the other hand, the concept of SPLs offers unique opportunities for the rapid development and deployment of certain services provided by local governments. Actually, software product lines can help to rapidly develop families of services, reducing costs and development efforts, as well as facilitating service integration. Thus, integrating both approaches seems the way to go. Various formalisms exist to formally model and verify SPLs. Here we concentrate on Featured Timed Automata due to recurrent time requirements in the e-government domain.

III. Background

This section presents some technical background on Software Product Lines, Featured Timed Automata, and UPPAAL.

An SPL is a set of software systems that share a high number of *features* while differ on others, where concrete configurations are derived from a core of common assets in a prescribe way. A feature it is usually referred as a characteristic or behavior of the system visible to the user. It captures both requirements of the end users as well as implementation concepts. For example, in the case study, some public administrations may require the payment of license applications. A public license service that requires a payment may offer credit card payments (cc), PayPal (pp), or both. In this case, cc and pp are features of the SPL that represent the presence of credit card and PayPal payments, respectively, while licensing services with credit card payments, PayPal payments, both, or none, are different services that can be derived from the family.

The variability of the SPL is defined in terms of common and variable features, usually through *feature models*. A future model expresses the valid combination of features, where each combination is a product in the family. In the previous example, the valid products would be given by the feature model $fm = \{\{\}, \{cc\}, \{pp\}, \{cc, pp\}\}$, where the empty set represents base functionality.

Featured Timed Automata is an extension to Timed Automata (TA) introduced by [1] for modeling families of TA. This is achieved by associating Boolean expressions over a set of features, to transitions (edges), referred to as *feature expressions*. Given a selection of features FS, it is possible to project a FTA into a TA. Intuitively, a feature expression associated with a transition indicates the latter must be present in all products that satisfy the former.

As in TA, a clock c models continuous and dense-time, it can only be inspected or reset to zero, and represents the time elapsed since its last reset. All clocks of a FTA are incremented synchronously as the automata evolves. A *clock valuation* η for a set of clocks C is a function $\eta : C \to \mathbb{R}_{\geq 0}$ that assigns each clock $c \in C$ to its current value ηc. A *clock constraint* is a logic condition over the value of a clock. Formally, feature expressions, clock constraints, and their satisfiability, are defined as follows [1].

A feature expression φ is a Boolean expression over a set of features F is defined as follows

$$\varphi ::= f \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg \varphi \mid \top$$

where $f \in F$ is a feature. We use Φ_F to denote the set of all possible feature expressions over a set of features F.

A *clock constraint* over a set of clocks C, written $g \in CC(C)$ is defined as follows

$$g ::= c < n \mid c \leq n \mid c > n \mid c \geq n \mid g \wedge g \mid \top$$

where $c \in C$, and $n \in \mathbb{N}$.

Given a feature selection $FS \subseteq F$ over a set of features F, and a feature expression $\varphi \in \Phi_F$, FS satisfies φ, noted $FS \models \varphi$, if

$$\begin{array}{lll} FS \models \top & \textit{always} & \\ FS \models f & \iff & f \in FS \\ FS \models \varphi_1 \diamondsuit \varphi_2 & \iff & FS \models \varphi_1 \diamondsuit FS \models \varphi_2 \\ FS \models \neg \varphi & \iff & FS \not\models \varphi \end{array}$$

where $\diamondsuit \in \{\wedge, \vee\}$.

The *satisfaction* of a clock constraint g by a clock valuation η, written $\eta \models g$, is defined as follows

$$\begin{array}{lr} \eta \models \top & \text{always} \\ \eta \models c \,\Box\, n & \text{if } \eta(c) \,\Box\, n \\ \eta \models g_1 \wedge g_2 & \text{if } \eta \models g_1 \wedge \eta \models g_2 \end{array}$$

where $\Box \in \{<, \leq, >, \geq\}$.

Definition 1: A feature timed automaton, is a tuple $\mathcal{A} = (L, L_0, A, C, T, I, F, fm, \gamma)$ where L is a set of locations, $L_0 \subseteq L$ is the set of initial locations, A is a set of actions, C is a set of clocks, $T \subseteq L \times CC(C) \times A \times \mathbf{2}^C \times L$ is a

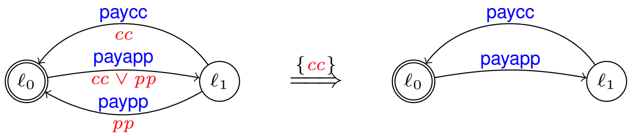


Fig. 1: Example of a FTA over features cc and pp (left), and its projection over the feature selection $\{cc\}$ (right).

set of transitions, $I\colon L \to CC(C)$ is the invariant, a partial function that assigns clock constraints to locations, F is a set of features, $fm \subseteq \mathbf{2}^F$ is a feature model, and $\gamma\colon T \to \Phi_F$ is a total function that assigns feature expressions to transitions.

Figure 1 (left) shows a FTA for the example introduced before and its projection into a TA by selecting the set of features $\{cc\}$ (right). A payment for an application (action payapp) can be requested only if the system supports payments by credit card or by PayPal, indicated by the feature expression $cc \vee pp$. Once the payment is requested, the system moves to a new state (ℓ_1) and waits for the user to select one of the available methods. When the FTA is projected onto the feature selection $FS = \{cc\}$, only the set of transitions with feature expressions that are satisfied by FS remain in the model.

UPPAAL is a real-time model checker based on TA theory. A system in UPPAAL is a network of parallel TA, where automata can transition independently or together by synchronizing over shared actions called channels. A transition with a channel a can only be taken when its dual channel in a neighbor automata is also in an transition that can be taken simultaneously. A channel and its dual channel are represented by a! and a?.

In addition, UPPAAL's modeling language enhanced the modeling of TA with integer variables, constants, arrays, user defined functions, and urgency, among others. Urgency can be represented in various ways, such as committed and urgent locations, and urgent channels. A *committed location*, marked with a letter C, is a location in which time can not pass, thus when an automaton in a network is in a committed location, only outgoing transition from such location are enabled. An *urgent location*, marked with a letter U, is a location that should be left without delay. Committed locations are more restricted than urgent locations. An *urgent channel* is a synchronization action that should be executed as soon as it is enabled by the transitions' guard.

IV. Case Study

The case study is based on two real cases of licensing public bus passenger services from Portugal and Ireland. Three types of licenses are required for the provision of such services: 1) a license to operate passenger services, 2) a license to provide a public bus passenger along a route, and 3) a license for each vehicle to transport passengers. All services in the family support submissions and assessment of licensing requests. Some licensing services, in addition, require payment before submitting a request, others allow appeals on rejected requests (apl), or both. Furthermore, services that require payment support different payment methods, namely credit cards (cc), PayPal payments (pp), or both. Functionality is divided in components and provided as follows.

Application – applicants submit the required documents (subdocs), pay a fee (payapp) if features pp or cc are present, and submit the application. Authorities can accept, consider incomplete (missing required documents), or reject the request. If the request is accepted or incomplete, it is closed, i.e., no further actions can be taken. If it is rejected and it is not possible to appeal ($\neg apl$), the request is closed. If feature apl is present, the applicant has 31 days to appeal, otherwise the request is closed as rejected. If an appeal is submitted, it can be rejected or accepted, and the request is closed.

Queue – as requests are submitted, they are queue and assigned to a particular authority as soon as they are available.

Authority – each authority is responsible for various requests. For each request that arrives, it has to preassess if all required documents are present, in which case it proceeds to assess the request.

PreProcessing – the authority has 31 days to evaluate if all documents are present and proceed to the assessment of the request, or notify the applicant if they are incomplete, closing the application (closeApp).

Processing – the authority has 90 days to evaluate the application and make a decision of whether accept it or reject it. If it is accepted the application is closed. If it is rejected, and feature apl is supported, the authority waits 31 days for an appeal. If it is submitted, it has 60 days to assess the new evidence, make a decision, and close the application. If the applicant does not appeal within the time or if apl is not supported, the application is closed.

CreditCard (*CC*) and *PayPal* (*PP*) – these are external payment services. If required, a user can pay fees by credit card (paycc) or PayPal (paypp). After initiated, the user has 1 day to proceed with the payment which can result in success (paidcc and paidpp) or cancellation (cancelcc and cancelpp).

A. Modeling Behavior

The UPPAAL automata modeling the components described before and some additional automata to orchestrate their interactions are shown in Figure 2. Below we describe some modeling decisions.

First, in order to model families of TA in UPPAAL, features are encoded as boolean variables, and feature expressions associated with transitions are encoded as logic guards over such boolean variables. Second, since various applications can be submitted concurrently, it is necessary to model that each application has its own *PreProcessing*, *Processing*, and payment related processes (*PP*, *CC*, *selectPayement*, *mergePaid*, and *mergeCancelPay*). Thus, some synchronization channels are declared as an array of channels, indexed by the application id, ranging from 0 to the maximum number of applications, represented by a global constant APPS. Third, a global constant AUTH indicates the number of authorities. Finally, there is only one *Queue* that receives applications.

Application – an application starts in a location apply and it can finish in four states: payment_cancelled, inclomplete_app, accepted, or rejected. *Urgent locations* ensure applications will be submitted. The submit channel is the only one not indexed by the application's id. This is because this channel synchronizes with the queue automaton and there is only one queue, otherwise, UPPAAL would create an automaton queue for each possible application. A global variable ca is used to hold the id of the application currently submitting, and it is passed by reference to *Application*, *Queue*, and *Authority*. Thus, the transitions labelled with submit in the application's automaton assign the application's id to such variable. In order to reduce the search space during verification, applications are submitted in order of id. In order to submit, an application has a guard ready() to check if all applications with lower id have submitted. When submitting, each application sets to true a global Boolean array nextapp in its corresponding entry. This is also done if a payment is canceled, since the application is no longer active. Two clocks, *tproc* and *tapl*, track the elapsed time since submission, and the appeal window time, respectively. An invariant in location can_appeal controls that an appeal must be made within 31 days, or the application is closed.

Queue – when receiving a submission (submit), the corresponding id, is added at the end of the queue – id is a parameter by reference bounded to ca. A variable len holds the length of the queue. As soon as the queue is not empty, it informs authorities that there are pending applications, pendingApps (urgent channel), and uses ca to indicate which application should be dequeue. The function front() returns the front of the queue. When an authority retrieves an application, getApp, it is removed by dequeue().

Authority – in order to balance the work load, a global array maintains the authorities current work load. An authority can open a new application if it is the authority with less work, controlled by the guard canOpenApp(), in which case, immediately retrieves the application from the queue, adds it to a list of current opened applications, newApp(id), and starts a pre-processing process for that application, preassess[id] – id is a parameter by reference bounded to ca. These three actions are done atomically, as indicated by the committed states between actions. When an application is closed, closeApp[id], only the authority that opened such application can close it, controlled by inOpenApps(id), removing the application from the current opened list using removeApp(id).

PreProcessing – a clock *tpreproc* tracks the pre-assessment time, which should not exceed 30 days, as indicated by the corresponding invariant. If documents are incomplete, ca is set with the application id, enabling the authority to closes the corresponding application.

Processing – clocks *tproc*, *tapl*, and *taplproc* track the assessment, appeal, and appeal processing time, which should not exceed 90, 32, and 60 days, respectively, as indicated by the corresponding location invariants. After a decision is made, ca is set with the application id, enabling the authority to closes the corresponding application.

PP and *CC* – both automata are symmetric. Clocks, tpp and tcc, track the elapsed payment time, which should not exceed one day, as indicated by the corresponding invariant.

In addition, we model automata to orchestrate the way the previous automata interact. Their functionality is given as follows. *selectPayment* – synchronizes payment requests (payapp) with payment by credit card or PayPal (paypp or paycc). This automata corresponds to the FTA of Figure 1. *mergePaid* and *mergeCancelPay* – synchronize a successful or cancellation payment response from either *PP* or *CC* (paidpp and paidcc, and cancelpp and cancelcc), respectively, and notify the applicant (paidapp or cancelpay).

The parallel composition of these models is a network of FTA, modeled in UPPAAL, representing a SPL with feature model $fm = \{\{\}, \{cc\}, \{pp\}, \{apl\}, \{cc, pp\}, \{cc, apl\}, \{pp, apl\}, \{cc, pp, apl\}\}$.

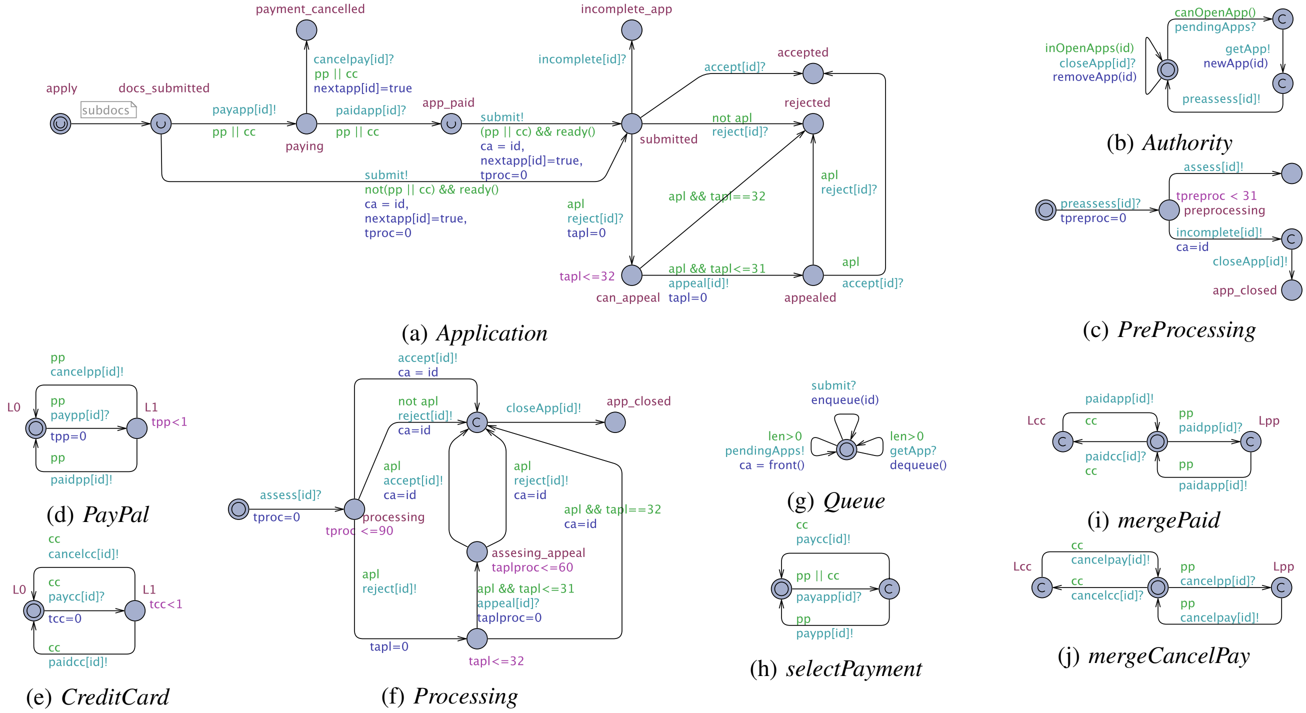
B. Verifying Properties

In order to verify temporal properties of FTA in *Uppaal*, it is necessary to model valid products. This can be done by adding an automaton to the network to represent the feature model. Such an automaton consists of an initial committed location and as many outgoing transitions to new locations (one per transition) as valid feature selections. Each transition assigns the Boolean value true to each variable that corresponds to a feature contained in the corresponding feature selection. The committed state ensures that a feature selection is made before any other transition is taken. However, this is not the case if there exists another automaton with initial committed states. Such situation should be avoided. The feature model of the SPL described befored is shown in Figure 3.

Examples of properties that can be verified in UPPAAL are listed in Table I and can be interpreted as follows: P1 – an application will eventually result in an accepted, rejected, incomplete or canceled application, where ap0 represents an application with id 0. This property is verified for all ids in [0,APPS-1]; P2 – if either a PayPal or credit card payment is canceled, the application eventually will be canceled. The property corresponds to application 0 but can be proved for all applications as before; P3 – if cc is not present, it is not possible to eventually be in a state related to credit card payments.; P4 and P5 – a submission and an appeal are processed within 121 days and 60 days after being submitted, respectively; P6 – it is not possible for an application to be opened by more that one authority at the same time. Here auth0 and auth1 are the only authorities. These properties have been checked for APPS = 4, and AUTH = 2. Due to the complexity of the model in terms of shared variables and features, increasing the number of applications to 5 hinders the verification due to the quick explosion in the state space.

V. Discussion

By using formalisms like FTA it is possible to take advantage of the commonalities present in all systems and define variations of family members over the common set of features. In addition the formal nature of FTA allows to verify whether



(a) *Application*

(b) *Authority*

(c) *PreProcessing*

(d) *PayPal*

(e) *CreditCard*

(f) *Processing*

(g) *Queue*

(h) *selectPayment*

(i) *mergePaid*

(j) *mergeCancelPay*

Fig. 2: UPPAAL automata with FTA variability modeling domain functionality.

TABLE I: Examples of verifiable temporal properties in Uppaal (using Uppaal's syntax).

Property		Ref
Liveness	`ap0.apply --> (ap0.accepted || ap0.incomplete_app || ap0.payment_cancelled || ap0.rejected)`	P1
	`(mergeCancelPay(0).Lpp || mergeCancelPay(0).Lcc) --> ap0.payment_cancelled`	P2
Reachability	`!cc --> !(exists(i:app_id) (CreditCard(i).L1 || mergeCancelPay(i).Lcc || mergePaid(i).Lcc))`	P3
Safety	`A[] ap0.submitted imply ap0.tproc <= 90+31`	P4
	`A[] ap3.appealed imply ap3.tapl <=60`	P5
	`A[] forall(i:app_id) !(auth0.inOpenApps(i) && auth1.inOpenApps(i))`	P6

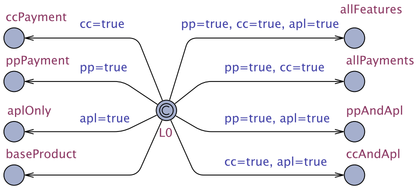


Fig. 3: UPPAAL automaton modeling the *feature model*.

the models satisfy a given property using well known real-time model checkers, such as UPPAAL. However, despite advantages of FTA, as we continued to refine the models of the case study discussed, we notice that further support is needed for more modular and compositional modeling of families of services. We propose a simple scenario to illustrate how orchestrating the interaction of FTA with variable orchestration mechanism, i.e., depending on the presence of features, can become cumbersome and error prone.

In alignment with the *only-once principle* mentioned in Section II, a typical scenario is the ability to consult relevant agencies whether a given applicant possesses a criminal record or has all tax duties in order. We model such external sources as FTA in Figure 4 (left). The model on the top left represents an external database that receives a request to check a given tax number (checkTax) and provides a response certifying whether tax duties are in order (resTax), while the model on the bottom left represents an external database that receives a request to check a person ID (checkCR) and provides a response certifying whether the person has criminal records (resCR). Their presence depends on features tx and cr, respectively.

In this scenario, the authority must consult the required external sources, if available, and wait for their responses before deciding to grant the license. Figure 4 (right) shows such an FTA, which is a simplification of automata *Processing* from Figure 2. When an application is ready to be assessed (assess), the authority consults external sources (checkES) if supported by the service, waits until all responses are ready (results), and makes a decision. In case the service does not support consultation of external sources the authority can directly make a decision.

The complication arise when modeling the interaction between the new FTA *Assess* and the external databases. First,

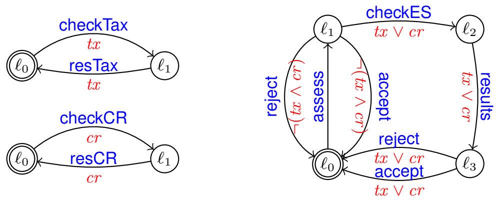


Fig. 4: Two FTA modeling external databases (left) and an FTA modeling an *Assess* component.

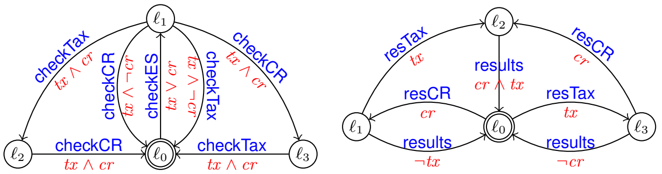


Fig. 5: Variable orchestration mechanisms, invoking two databases (left), and waiting for their responses (right).

it is necessary to model the possibility of invoking both databases in any possible order, but considering that the presence of both databases is optional, thus all combinations of presence of features tx and cr must be considered. The FTA on the left of Figure 5 illustrates such a model. The FTA waits for an action checkES, which will be present if at least one of the databases is supported ($tx \vee cr$). If only one database is present, it calls the corresponding database. If both are present, they can be called in any order. The difficulty in modeling such a mechanism is twofold: 1) modeling correctly the feature expressions of all transitions to avoid erroneous behavior, either coming from transitions that can never take place, or from transitions that can take place when they should not; and 2) correctly modeling all possible cases. For example, if the feature expression associated with transition $\ell_1 \xrightarrow{\text{checkTax}} \ell_0$ is erroneously set to $tx \wedge cr$ instead of $tx \wedge \neg cr$, the FTA allows calling only the database for taxes when both databases should be called. Furthermore, if in the future a new database is needed, for example to check if an applicant has a driving license in order, it will be necessary to redesign the entire FTA and calculate manually the new feature expressions. Such an automaton for three databases, has 11 locations and 25 transitions. Thus, in addition to time consuming, it can become cumbersome and error prone quickly. Similarly, the second step is to model that results may be ready in any possible order, and that the presence of a given result depends on the presence of the databases. The FTA on the right of Figure 5 illustrates such a model. The same potential issues arise in this example. As before, adding a new database results in an automaton of 16 locations and 30 transitions.

In addition, it is possible to recognize that these automata behave as a kind of variable *replicator* and a *join* orchestration mechanism, respectively. A typical replicator consisting of an input and two outputs receives an input signal and replicates it to its outputs, while a typical join consisting of two

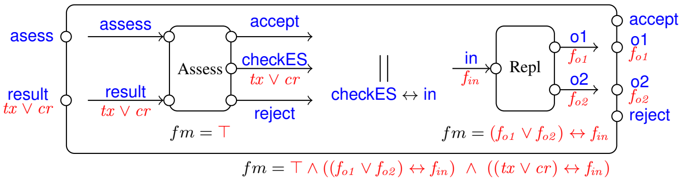


Fig. 6: An scheme of an enriched FTA enabling inference of variability restrictions among orchestrated automata.

inputs and one output waits for both input signals before emitting an output signal. Given that these sort of orchestration patterns occur recurrently when modeling families of services, it is interesting consider to enrich FTA in order to facilitate the specification of generic orchestration mechanisms to orchestrate services, similar to how connectors in the Reo [3] coordination language orchestrate components. Such an extension should enable the creation of complex orchestration process by means of composition, without the need to calculate the variability in the transitions. Instead, variability would be inferred by the context, namely, the variability of the services to orchestrate. In addition, by explicitly separating the modeling of orchestration mechanism from the rest of the systems' functionality, it would be possible to model modular FTA. This facilitates the maintainability of the SPL model, for example, by enabling faster replacement or extension of specific functionality or the way they are orchestrated.

In the envision extension, FTA are treated as components with explicit interfaces, namely the actions on which they synchronized. Interfaces can be input or output actions. Each interface action in a given automata has a variability associated, which can be inferred from the variability associated with transitions labeled with such actions, i.e., an interface is present if at least one transition labeled with such an action is present. Each automaton expresses its feature model. Thus, an enriched composition mechanism, would enable to compose automata by explicitly linking interfaces, thus avoiding the need to use identical action names when modeling automata. The composition would compose their feature models, and impose additional restrictions over the interfaces linked, i.e., linked interfaces in addition to depending on their variability would depend on each other. Thus, orchestration mechanism can be modeled using generic names for actions and feature expressions, they can have their own feature model, and when composed, their interfaces would depend, in addition, on the actual interfaces they connect to.

Figure 6 illustrates this concept. *Assess* corresponds to the FTA shown in Figure 4, with $fm = \top$ to indicate that any combination of features cr and tx is possible. *Repl* is a generic version of the *Repl* in Figure 5 modeled in terms of features f_{in}, f_{o1}, and f_{o2}. The associated feature model indicates that an output interface can be present, if and only if, the input is present. By composing both FTA and linking interfaces checkEs and in, the variability of the resulting automata is inferred from the feature model of both automata and the restriction imposed by the new connection. In the resulting automata interfaces checkES and in are hidden, given that they become internal actions.

Adding a new database to the previous example would reduce to replace the current *Join* and *Repl* by a *Join* of three inputs and one output, and a *Repl* of one input and three outputs, respectively. For example, the three input join can be created by connecting the output of a generic join of two inputs, to an input of another generic join of two inputs.

VI. Related Work

This section provides a brief overview of related work regarding formal behavioral modeling and verification of SPL, development of electronic licensing services, and use of SPL to support e-government.

Regarding formal modeling and verification of SPL relevant approaches in the literature include the following. In [4] the authors proposed Featured Transitions Systems to model families of Transitions Systems. The semantics of Featured Timed Automata is based on such an extension. In [1] the authors of FTA also introduce *ProVeLines*[2], a product line of verifiers for software product lines. The advantage of using this tool is that it enables to identify all products that do not satisfy a given property. However, the installation and use of the tool is not straightforward. In [5] the authors propose an extension to Petri Nets, called Feature Nets (FNs) to specify the behavior of a SPL in a single model.

Regarding the development of electronic licensing services, only a few relevant studies were found in the literature. In [6], the authors propose a composite domain framework for rapid development of electronic public services (EPS). It includes frameworks for building the front office and back office part of an EPS. A software infrastructure and a software process is proposed in [7] for the rapid development of EPS and its application is shown in [8] through a case study focused on delivering licensing services. In [9], the authors propose an interoperability integration framework to align the organizational structures and processes of different government agencies and to provide integrated public services.

Regarding SPL support for e-government, only a few studies were found in the literature. In [10] the authors propose a software product line for generating front-end environments for an e-government context management system. In [11] the authors propose a method to generate personalized government documents using SPL. The approach takes advantage of the high level of reuse in government documents.

VII. Conclusions

In this paper we try to motivate the use of formal methods and software product lines for the e-government domain. We presented a case study on modeling families of public licensing services by using the FTA formalism and UPPAAL model checker, and analyze the suitability of such an approach for this domain. Although FTA is a rich formalism, we believe it can be extended to provide better support in the modeling of distributed services. In particular, the design of orchestration mechanism for families of distributed services can become cumbersome and error prone using bare FTA. We illustrate this with a scenario from the domain and provide hints on how to extend FTA.

Based on the lessons learned from the case study, we are currently developing such an extension to FTA, which can simplify the modeling of families of service by means of composition.

Acknowledgment

This work is part of the project SmartEGOV: Harnessing EGOV for Smart Governance (Foundations, Methods, Tools) / NORTE-01-0145-FEDER-000037, supported by Norte Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, through the European Regional Development Fund (ERDF). The first author is further supported by FCT under grant PD/BD/52238/2013.

References

[1] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay, "Behavioural modelling and verification of real-time software product lines," in *Proceedings of the 16th International Software Product Line Conference-Volume 1*. ACM, 2012, pp. 66–75.

[2] L. Veiga, T. Janowski, and L. S. Barbosa, "Digital government and administrative burden reduction," in *Proceedings of the 9th International Conference on Theory and Practice of Electronic Governance, ICEGOV 2016, Montevideo, Uruguay, March 1-3, 2016*, 2016, pp. 323–326. [Online]. Available: http://doi.acm.org/10.1145/2910019.2910107

[3] F. Arbab, "Reo: a channel-based coordination model for component composition," *Mathematical Structures in Computer Science*, no. 3, pp. 329–366, 2004.

[4] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 335–344.

[5] R. Muschevici, J. Proença, and D. Clarke, "Feature nets: behavioural modelling of software product lines," *Software & Systems Modeling*, pp. 1–26, 2015.

[6] A. K. Ojo, T. Janowski, and E. Estevez, "A composite domain framework for developing electronic public services." in *International Conference on Software Engineering Theory and Practice (SETP)*, 2007, pp. 234–241.

[7] T. Janowski, A. Ojo, and E. Estevez, "Rapid development of electronic public services: Software infrastructure and software process," in *Proceedings of the 8th annual international conference on Digital government research: bridging disciplines & domains*. Digital Government Society of North America, 2007, pp. 294–295.

[8] ——, "Rapid development of electronic public services: A case study in electronic licensing service," in *Proceedings of the 8th annual international conference on Digital government research: bridging disciplines & domains*. Digital Government Society of North America, 2007, pp. 292–293.

[9] M. Al-Husban and C. Adams, "Connected services delivery framework: Towards interoperable government," *Emerging Mobile and Web 2.0 Technologies for Connected E-Government*, vol. 50, 2014.

[10] V. M. A. de Lima, R. M. Marcacini, M. H. P. Lima, M. I. Cagnin, and M. A. S. Turine, "A generation environment for front-end layer in e-government content management systems," in *Web Congress (LA-WEB), 2014 9th Latin American*. IEEE, 2014, pp. 119–123.

[11] M. C. Penadés, P. Martí, J. H. Canós, and A. Gómez, "Product line-based customization of e-government documents," in *PEGOV 2014: Personalization in e-Government Services, Data and Applications*, vol. 1181. CEUR-WS, 2014.

[2] https://projects.info.unamur.be/fts/provelines/