Orchestration of Dynamic Service Product Lines with Featured Modal Contract Automata

Davide Basile
University of Florence, Italy
ISTI–CNR, Pisa, Italy
d.basile@isti.cnr.it

Maurice H. ter Beek
ISTI–CNR, Pisa, Italy
m.terbeek@isti.cnr.it

Felicita Di Giandomenico
ISTI–CNR, Pisa, Italy
f.digiandomenico@isti.cnr.it

Stefania Gnesi
ISTI–CNR, Pisa, Italy
s.gnesi@isti.cnr.it

ABSTRACT

In Service-Oriented Computing, contracts provide a way to characterise the behavioural conformance of a composition of services, and to guarantee that the results do not lead to spurious compositions. Adding variability leads to a product line of services capable of adapting to customer requirements and to changes in the context in which they operate. To this aim, we extended a previously introduced formal model of service contracts. In particular, we included: (i) feature-based constraints and (ii) four classes of service requests to characterise different variants of service agreement. We then exploited Supervisory Control Theory to define an algorithm to synthesise an orchestration of services that satisfies: (i) all feature constraints of the service product line, and (ii) the maximal number of service requests for which an agreement can be reached. Moreover, such an orchestration of a service product line, whose number of products is potentially exponential in the number of features, can be synthesised from only a subset of its products. A prototypical tool supports the developed theory. In this short paper, we provide the intuition for our approach and illustrate it by means of a Hotel reservation service product line.

CCS CONCEPTS

•Information systems → Service discovery and interfaces;
•Software and its engineering → Software product lines;

KEYWORDS

Services, Product lines, Featured Modal Contract Automata

ACM Reference format:
Davide Basile, Maurice H. ter Beek, Felicita Di Giandomenico, and Stefania Gnesi. 2017. Orchestration of Dynamic Service Product Lines with Featured Modal Contract Automata. In *Proceedings of SPLC '17, Sevilla, Spain, September 25-29, 2017,* 6 pages.
DOI: 10.1145/3109729.3109741

1 INTRODUCTION

Service-oriented computing (SOC) [20] is a paradigm for distributed computing based on the publication, discovery and orchestration of *services.* Web applications reuse services in different configurations over time, e.g. due to the need to adapt to changes in the environment or to the resources of the devices on which they run. Therefore the idea to organise them into *dynamic service product lines* (DSPL) was first explored a decade ago in the SOAPL workshop series at three consecutive SPLC conferences (cf., e.g., [22, 26, 27]), followed by applications for Web stores, smart grids, services as used in grid computing and e-Government public licensing services [1, 12, 18].

On a different line, *service contracts* [3] have been introduced to formally describe the behaviour of services in terms of their obligations (i.e. *offers* of the service) and their requirements (i.e. *requests* by the service). Contracts characterise an *agreement* among services as an orchestration (i.e. a composition) of them based on the satisfaction of all requirements through obligations. Orchestrations can dynamically adapt to the discovery of new services, to service updates and to services that are no longer available.

In [5], *contract automata* were introduced as a formal model for service contracts. They represent either single services (called *principals*) or compositions of services based on orchestrated or choreographed coordination [6]. The goal of each principal is to reach an accepting (final) state by matching its requests with corresponding offers of other principals. Through service contracts it is then possible to characterise the behaviour of an ensemble of services. The notion of *agreement* then characterises safe executions of services (i.e. all requests matched with corresponding offers).

In [8], contract automata were equipped with variability mechanisms to distinguish *necessary* (□) from *permitted* (◇) requests, mimicking uncontrollable and controllable actions, respectively, from Supervisory Control Theory (SCT) [16]. Offers were only permitted as dictated by agreement. Contract agreement guaranteed the fulfilment of all necessary requests and negotiated the maximum number of permitted requests that could be fulfilled without spoiling the service composition. However, none of the above formalisms natively support product line modelling.

In this paper, we briefly introduce *featured modal contract automata* (FMCA) for modelling contract-based DSPL. These FMCA extend the aforementioned modal service contract automata (MSCA) from [8] with the possibility to define: (i) *feature constraints* on service actions (requests and offers); and (ii) *urgent*, *greedy* and *lazy* necessary service requests. Features are identified as service actions, and each FMCA represents a behavioural product line of services equipped with feature constraints. A feature constraint can be any of the constraints used in feature models, including cross-tree constraints, defined as its corresponding propositional formula (cf. [10, 25]). Each product is identified as a truth assignment satisfying the feature constraints.

Urgent, greedy and lazy requests are, in decreasing order of relevance, necessary service requests with further restrictions on their satisfiability. Similarly to [8], permitted requests are optional and can thus be discarded for reaching an agreement.

The main contributions of the approach illustrated in this short paper are as follows: (i) we informally describe a new formalism for modelling contract-based DSPL; (ii) an example shows the benefits of adopting the proposed approach; (iii) we discuss an algorithm for synthesising an orchestration of a service product line model that allows for dynamic adaptation. The result of the latter is the so-called *maximally permissive controller* (*mpc*) satisfying all feature constraints, all variants of necessary service requests and the maximal number of permitted requests.

A main feature of FMCA relies on the synthesis of an orchestration for a service product line model. Indeed, the number of products of a product line is in general exponential in the number of features, and the problem of generating an orchestration of services is to scale to larger number of features. To this aim, the synthesis algorithm of FMCA organises the products in a partial order. This allows to visit a potentially smaller set than all products, making the approach scale. This (partial) order relates each product to its sub- and super-products, i.e. those products in which more or less, respectively, variability has been resolved. Products in which not all variability has been resolved are also known as subfamilies.

Finally, the theory presented in this paper has been implemented in an open-source prototypical tool [7] (cf. Fig. 3), with a demo and the source available at https://github.com/davidebasile/FMCAT/, and its applicability is further demonstrated by a running example.

2 ILLUSTRATING THE APPROACH

We informally illustrate the use of FMCA with the help of an intuitive example. We consider a simple franchise of Hotel reservation systems and model it as a service product line. Such a system consists of clients interested in booking a room in a Hotel, which offers either credit card or cash payments and possibly emits an invoice according to the Hotel feature model depicted in Fig. 1a.

2.1 Feature model

A feature model defines all products of a product line and it has a corresponding propositional formula φ over (primitive) features, called *feature constraint*. We identify features as actions (requests and offers) performed by services. Moreover, each product is identified by its set of *required* (literals interpreted as *true* in φ) and *forbidden* features (literals interpreted as *false* in φ). All required features must be present in the service, while none of the forbidden features may be present.

For a lighter presentation in this short paper, we consider only the Hotel product line's feature model depicted in Fig. 1, i.e. we ignore the Client's feature models. Generally, when different service contracts specify different feature models, their composition takes the conjunction of the corresponding feature models. This paves the way to the specification of DSPL, by adding new features to the existing feature model through composition. Moreover, services and their feature models are composed at binding time. We remark that the approach does scale to larger numbers of features [9].

The Hotel's feature model allows two *alternative* payment methods: *cash* or *card.* Moreover, any product offering cash payment, *requires* the *invoice* feature to be present. Indeed, the Hotel franchise (i.e. the product line) wants to prevent any of its hotels (i.e. a product) to perform off-book payments. Thus the feature constraint corresponding to the feature model of Fig. 1 is:

$$\varphi = ((card \land \neg cash) \lor (cash \land \neg card)) \land (\neg cash \lor invoice)$$

The three *products* defined by the feature model are depicted in Fig. 1, together with a super-product p_1 in which the presence of the *invoice* feature has not yet been resolved (unresolved features will be activated by the orchestration, if possible). These are the products satisfying φ (e.g. $card = true$ and $cash = false$ satisfies φ, denoted by $\varphi \models_{p_1} true$). Products can be ordered according to their required and forbidden actions. In Fig. 1, p_2 and p_3 are subproducts of p_1, written $p_2 \preceq p_1$ and $p_3 \preceq p_1$. Indeed, the required and forbidden actions of p_1 are contained in those of p_2 and p_3. This ordering is exploited for efficiently verifying all products.

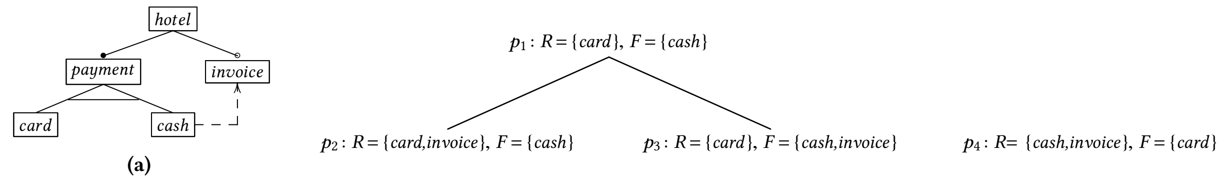
2.2 Behavioural contracts

A service contract characterises service behaviour in terms of offer and request actions, drawn respectively as overlined and non-overlined labels, while permitted transitions are depicted as dotted (cf. Fig. 2). We extend contracts from [8] by indicating "when" (i.e. in which states) necessary requests have to be matched. Therefore, we partition the set of necessary service requests into urgent ($\Box_u$), greedy ($\Box_g$) and lazy ($\Box_\ell$) requests. Urgent requests are the most restrictive and must be matched whenever they can be executed. Greedy requests must be matched as soon as possible, i.e. their execution can be delayed until a match is available. Lazy requests are the least restrictive and only require to be matched somewhere.

In the Hotel reservation service scenario, we assume two classes of Clients: business and economy. In Fig. 2a, the contract of a BusinessClientG is depicted. It starts by requiring to book a room ($room\Box_u$). The Client request in this case is *urgent*, due to its business priority. Once the room is selected, the client can either perform an off-book cash-only payment ($\overline{cash}$), not requiring any receipt or invoice, or a credit card payment ($\overline{card}$). Assuming a client travelling for business, an invoice or receipt is needed for being reimbursed by the clients' organisation. The organisation must accept invoices, while receipts may be rejected.

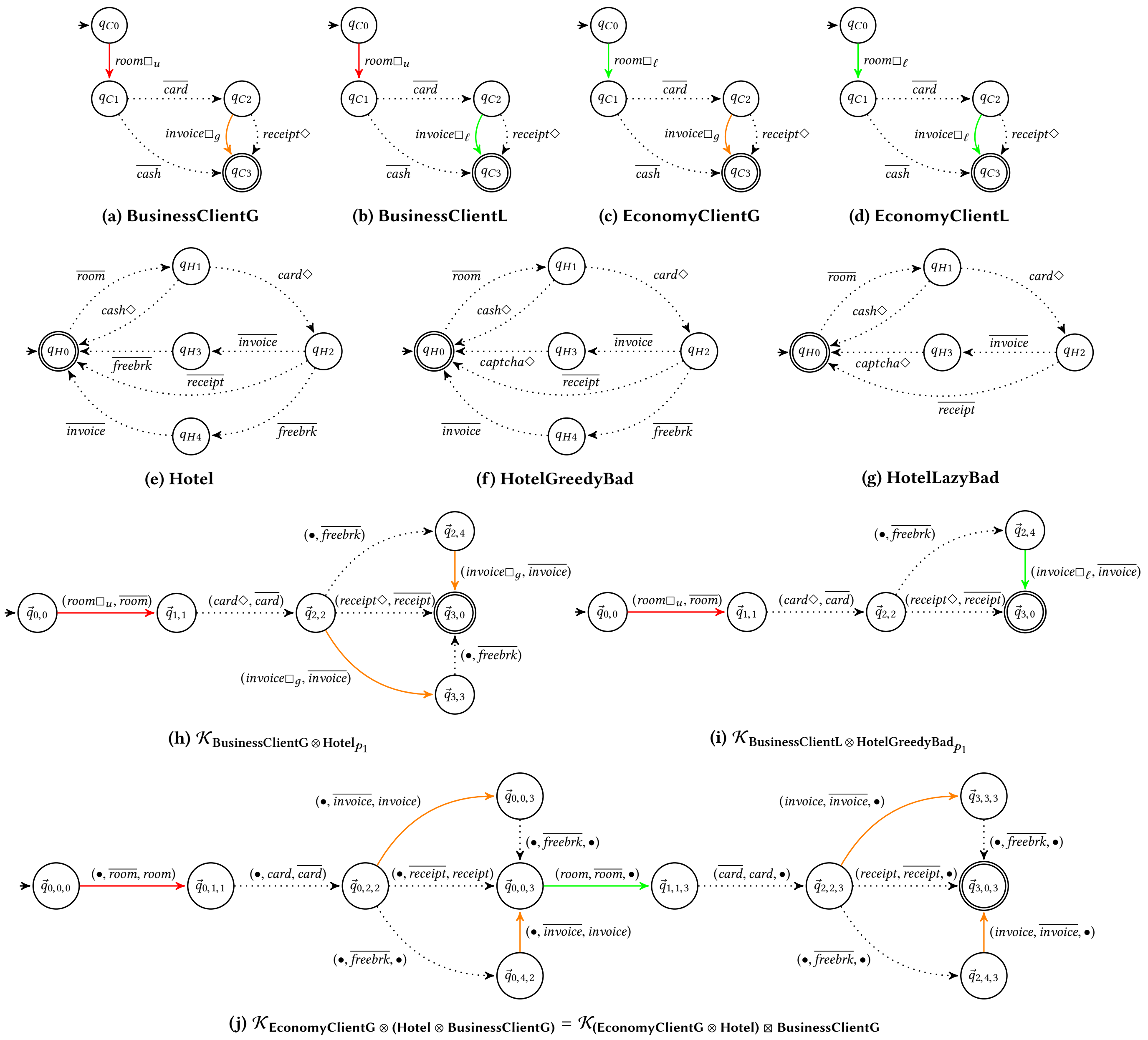
In case of cash payments, a client is (maliciously) using false invoices to ask larger reimbursement sums (e.g. a hotel could be owned by an accomplice). In case of (honest) payment by credit card, the client will require an invoice ($invoice\Box_g$) or a receipt ($receipt\Diamond$) from the hotel. The invoice request is marked as (necessary) greedy.

The contract of an EconomyClientG is similar to that of a BusinessClientG and it is depicted in Fig. 2c. In particular, its room request is marked as lazy ($room\Box_\ell$), i.e. with a lower priority. We will also consider a "lazier" version of both clients, depicted in Fig. 2b and in Fig. 2d. In these contracts, the invoice request is marked as lazy ($invoice\Box_\ell$), with ClientG and ClientL indicating the greedy and lazy version, respectively, of the invoice request.



hotel
payment
invoice
card
cash
(a)
p_1: R = {card}, F = {cash}
p_2: R = {card,invoice}, F = {cash}
p_3: R = {card}, F = {cash,invoice}
p_4: R= {cash,invoice}, F = {card}

Figure 1: The feature model and the partial order of products (with R = *Required* and F = *Forbidden*) of the Hotel product line



(a) BusinessClientG
(b) BusinessClientL
(c) EconomyClientG
(d) EconomyClientL
(e) Hotel
(f) HotelGreedyBad
(g) HotelLazyBad
(h) $\mathcal{K}_{\text{BusinessClientG} \otimes \text{Hotel}_{p_1}}$
(i) $\mathcal{K}_{\text{BusinessClientL} \otimes \text{HotelGreedyBad}_{p_1}}$
(j) $\mathcal{K}_{\text{EconomyClientG} \otimes (\text{Hotel} \otimes \text{BusinessClientG})} = \mathcal{K}_{(\text{EconomyClientG} \otimes \text{Hotel}) \boxtimes \text{BusinessClientG}}$

Figure 2: The Hotel reservation service product line

In Fig. 2e, the Hotel contract is depicted. Recall its feature constraint φ. The hotel service starts by offering a room ($\overline{room}$). It accepts payments by either credit card ($card\Diamond$) or cash ($cash\Diamond$). The mutual exclusion (xor) between these features (known to be not expressible by only an automaton [11]), specified in φ, ensures that exactly one type of payment is available in each Hotel product. In case of cash payments, the service returns to its initial (and final) state q_{H0}. Otherwise, it proceeds by emitting either a receipt

($\overline{receipt}$) or an invoice ($\overline{invoice}$). In the latter case, a free breakfast offer ($\overline{freebrk}$) is delivered as a gift before or after the invoice was emitted. After several such interactions, it returns to its initial state.

We also consider two different contracts for the Hotel service product line, called HotelGreedyBad (cf. Fig. 2f) and HotelLazyBad (cf. Fig. 2g), which share the same feature model but have a slightly different behaviour than Hotel. The HotelGreedyBad activates a captcha in case the clients select the offer $\overline{invoice}$ instead of the free breakfast offer, to avoid possible denial-of-service attacks. The HotelLazyBad is similar, except that it does not offer free breakfast and always performs a captcha check.

2.3 Composition of contracts

The FMCA composition operators are crucial for modelling DSPL, in particular for generating (at binding time) an ensemble of services. By adding new services to an existing composition, it is possible to dynamically update the service product line model and to synthesise, if possible, a composition satisfying all requirements defined by service contracts. Moreover, if newly added service contracts are equipped with different feature models, the overall composed feature model will be updated at runtime by adding (and possibly removing) new features to the model.

The FMCA formalism is endowed with three compositional operators: the product composition $\otimes$, the projection operator $\prod$ and the associative composition $\boxtimes$. Intuitively, product composition $\otimes$ interleaves the actions of all operands, with the only restriction that if two operands are ready to execute two complementary actions (i.e. a request matched by a corresponding offer) then only their match will be allowed and their interleaving prevented. The projection operator $\prod^i(\mathcal{A})$ retrieves the principal with index i involved in $\mathcal{A}$ and identifies its original transitions and feature constraint. The associative composition operator $\boxtimes$ is defined on top of the operators $\otimes$ and $\prod$. First, the corresponding principals of the operands are extracted by $\prod$ and then they are recomposed all together in a single step by $\otimes$. This causes all pre-existing matches to be rearranged. Their formalisation is available in [5, 9].

The sub-portion of the composition BusinessClientG $\otimes$ Hotel in agreement (i.e. orchestration) of product p_1 is depicted in Fig. 2h:[1] in this composition, the outgoing transition $\vec{q}_{0,0} \xrightarrow{(room\Box_u,\ \overline{room})}$ is an example of an urgent match between the urgent request $room\Box_u$ of the first principal (i.e. BusinessClientG) and the permitted offer $\overline{room}$ of the second (i.e. Hotel). Moreover, transitions $\vec{q}_{0,0} \xrightarrow{(room\Box_g,\ \bullet)}$ or $\vec{q}_{0,0} \xrightarrow{(\bullet,\ \overline{room})}$ are not allowed because $(room\Box_g, \bullet)$ and $(\bullet, \overline{room})$ are two complementary actions, thus removed in the composition.

In this case, the feature constraint (and hence the valid products) of the orchestration is exactly φ (recall that the client has no feature constraint). This orchestration is identical to the one of p_2 because the required invoice feature (required in p_2 and not in p_1) is available in both orchestrations. Conversely, the orchestration BusinessClientG $\otimes$ Hotel of products p_3 and p_4 is *empty*: no agreement exists. For product p_3, it forbids the necessary (greedy) invoice request, executable in both states $\vec{q}_{2,2}$ and $\vec{q}_{2,4}$.

While p_1, p_2 and p_3 are products featuring payments made by credit card, product p_4 corresponds to the Hotel product requiring payments by cash (and hence forbidding payments by credit card). In this case, the Hotel product line is protected from possible off-book payments by also requiring (via φ) the offer $\overline{invoice}$. The orchestration is indeed empty: the malicious behaviour is blocked.

Note here that requirements of products are stricter conditions than necessary requests. Indeed, the requirements of a product are defined on top of its behavioural contract, while necessary requests are defined inside the contract. Unreachable necessary requests do not spoil the contract agreement. Conversely, an unreachable action required by a product violates the contract agreement (e.g. $\overline{invoice}$ in p_4 is unreachable because $\overline{card}$ is forbidden).

Finally, the orchestration of the entire Hotel service product line is simply the orchestration of one of the products p_1 or p_2.

We now explain how the different classes of necessary requests affect the orchestration of service contracts. Moreover, below examples underline how the order in which services are added to the overall composition is crucial for obtaining a composition in agreement. We note that this order is a key aspect in DSPL, where new services must be added to the overall composition seamlessly, i.e. without corrupting the validity of the application. The considered orchestrations always refer to the entire product line model.

First we consider a composition of the Hotel service with both business and economy clients and show how urgent requests can be used to enforce priorities in service requests. If EconomyClientG is served *before* BusinessClientG, i.e. (EconomyClientG $\otimes$ Hotel) $\otimes$ BusinessClientG, then $t_1 = (\vec{q}_{0,0,0}, (room\Box_\ell, \overline{room}, \bullet), \vec{q}_{1,1,0})$, i.e. a match between lazy request and offer, is activated in the composition instead of transition $t_2 = (\vec{q}_{0,0,0}, (\bullet, \overline{room}, room\Box_u), \vec{q}_{0,1,1})$, i.e. an urgent match. As a result, the corresponding orchestration will be empty. Intuitively, the business client should be served before the economy client (i.e. t_2 instead of t_1). Indeed, an orchestration in agreement is admitted by EconomyClientG $\otimes$ (Hotel $\otimes$ BusinessClientG), with the business client served before the economy client (cf. Fig. 2j). Figure 3 depicts this orchestration as computed with FMCAT (cf. Sect. 3).

Next we consider the orchestration of the service composition BusinessClientG$\otimes$HotelGreedyBad, which is empty. This is caused by the request $(\vec{q}_{3,3}, (\bullet, captcha\Diamond), \vec{q}_{3,0})$ not matched (recall that

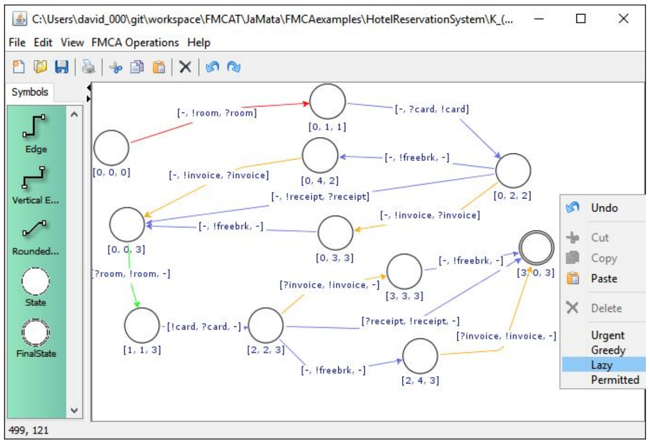


Figure 3: FMCAT with $\mathcal{K}_{\text{EconomyClientG}\otimes(\text{Hotel}\otimes\text{BusinessClientG})}$

[1] In Fig. 2, the subscripts of $\vec{q}$ identify the client's local state and the hotel's local state, in the order in which they are composed. Likewise for other compositions.

each request must be matched by a corresponding offer): the orchestration cannot prevent the execution of the greedy transition $(\vec{q}_{2,2}, (invoice\Box_g, \overline{invoice}), \vec{q}_{3,3})$. Indeed, the greedy invoice request of BusinessClientG requires to be matched as soon as possible: it cannot be "delayed" to the subsequent state $\vec{q}_{2,4}$ (cf. Fig. 2h).

On the other hand, the orchestration of the service composition BusinessClientL ⊗ HotelGreedyBad, depicted in Fig. 2i, is not empty. This is because in this case, the necessary lazy transition $(\vec{q}_{2,2}, (invoice\Box_\ell, \overline{invoice}), \vec{q}_{3,3})$, and consequently the permitted request transition $(\vec{q}_{3,3}, (\bullet, captcha\Diamond), \vec{q}_{3,0})$, are removed from the orchestration. This is possible since the necessary lazy request *invoice* of BusinessClientL is delayed to be matched in state $\vec{q}_{2,4}$.

Finally, consider the orchestration of the service composition BusinessClientL ⊗ HotelLazyBad. In this composition, all transitions incident in states $\vec{q}_{2,4}$ of Fig. 2h are absent (recall that HotelLazyBad does not offer free breakfast). The lazy match transition $(\vec{q}_{2,2}, (invoice\Box_\ell, \overline{invoice}), \vec{q}_{3,3})$ thus cannot be removed from the orchestration, as it is the only available match and the lazy invoice request is *necessary*. Also this orchestration is thus empty.

In the next section, we discuss the synthesis algorithm for computing the orchestration of services for a product line model.

3 COMPUTING THE ORCHESTRATION

In the previous section, several orchestrations have been described. Here, we will briefly describe the algorithm for synthesising an orchestration of FMCA, viz. the maximal sub-portion of an FMCA that is safe. The orchestration will be the *most permissive controller* (*mpc* for short) in the style of SCT for discrete event systems [16, 28]. A discrete event system is a finite state automaton, where *marked* (i.e. final) states represent the successful termination of a task, while *forbidden* states should never be traversed in "good" computations.

The purpose of SCT is to synthesise a controller that enforces good computations. To do so, it distinguishes between *controllable* events (those that the controller can disable) and *uncontrollable* events (those that are always enabled), besides partitioning events into *observable* and *unobservable* (obviously uncontrollable). If all events are observable, then an *mpc* exists which never blocks a good computation [16]. The purpose of contracts is to declare all executions of a principal in terms of requests and offers. Therefore, we assume that all actions of a (composed) contract are observable.

FMCA actions. Next we discuss when a transition of an FMCA is *controllable* or *uncontrollable*. The correspondence between permitted/necessary and controllable/uncontrollable was mainly exploited in [8], where all necessary requests were *greedy*. We added an extra layer of information about "when" a necessary request can be matched, mainly due to how we build the composition of FMCA (interleavings). All permitted actions are fully controllable. As briefly discussed previously, *urgent*, *greedy* and *lazy* requests have an increasing degree of controllability. An urgent request is fully uncontrollable: it must be matched in every possible state in which it can be executed. A greedy request can be disabled by the controller as long as the first match is available. Finally, a lazy request only requires to be matched: its matches are controllable by the orchestrator, provided at least one match is available.

The composition of contracts corresponds to the uncontrolled system (a.k.a. plant) in [16, 28]. Note that the composition of the *mpc* and the plant (i.e. controlled system) is not generated through the operators of composition of FMCA. As usual, the controlled system can be obtained by a standard synchronous composition of the *mpc* with the plant, which blocks all transitions that are in the plant but not in the *mpc*. The interactions between the orchestrator and the principals, that are used for realising the orchestration computed through the *mpc*, are implicit in our framework [6]. Clearly, the behaviour that we want to enforce upon a given FMCA corresponds exactly to the traces in agreement; thus we assume both (i) request transitions and (ii) forbidden transitions to lead to a forbidden state. To fulfil the modalities imposed by FMCA, we force a composition to be in agreement only if there exists a match for each necessary (urgent, greedy or lazy) request.

Moreover, we want to synthesise an orchestration of services that satisfies the feature constraint. To this aim, the synthesis algorithm computes the *mpc* of a *product* of the family identified by the featured constraint. Even though the number of products of a family is in general exponential in the number of features [15], through the partial order it is possible to synthesise the *mpc* for the entire product family from only a subset of valid products.

Orchestration synthesis. We now briefly describe the iterative algorithm for computing the *mpc* of product p of an FMCA $\mathcal{A}$. In [9] the algorithm is formally detailed. With respect to the standard synthesis in [28], we exploit non-local information related to other transitions for deciding whether a given transition is controllable or uncontrollable. At each step i, the algorithm updates incrementally a set of states R_i and revises an FMCA $\mathcal{K}_i$; it terminates when no more updates are possible. Intuitively, the property of agreement requires that all requests are matched. Hence, we want to remove all possible (non-matched) requests. We also want to remove all actions that are forbidden by the product. The *mpc* must prevent these "bad" transitions (requests and actions forbidden by the product) from being executed. This is straightforward for bad controllable transitions, while we can only try to make the bad uncontrollable transitions unreachable. To this aim, the sets R_i contain the "bad" states: those that cannot prevent a necessary request or a forbidden action to be eventually executed (i.e. states in uncontrollable disagreement). The algorithm terminates when no new updates are available. Upon termination, if the initial state is bad (in R_n) or some action required by product p is unavailable in $\mathcal{K}_n$, then the *mpc* is empty. Otherwise, the synthesised automaton $\mathcal{K}_n$ is the *mpc* of p. Since the set R_i is finite and can only increase in each step, the termination of the algorithm is guaranteed.

4 RELATED WORK AND CONCLUSION

In this short paper, we briefly illustrated FMCA as a novel formal model for expressing contract-based DSPL. We now compare FMCA with other formalisms from the literature and mention the benefits of adopting FMCA as a formal model for specifying DSPL.

The definition of FMCA builds on two automata-based models for modelling and analysing variability in product lines, both based on superimposing multiple product automata in a single, enriched family automaton. From Modal Transition Systems (MTS) [2], FMCA inherit the distinction into permitted (may) and necessary (must) transitions, whereas the explicit incorporation of feature constraints stems from Featured Transition Systems (FTS) [17].

MTS were first recognised as a suitable behavioural model for describing product lines in [19], which provided an algorithm to check the conformance of product behaviour against that of the product family. Subsequent extensions involving notions from interface automata and I/O automata were defined in [23] and [24], respectively. Another line of research led to MTS with an associated set of variability constraints expressed over actions and a dedicated variability model checker that allows one to verify a property for a family and conclude the result to hold for all its products [11, 13].

Compared to FMCA, none of these models can explicitly handle *dynamic* product lines, a characteristic FMCA inherits from [5, 6].

Furthermore, we tackled the problem of synthesising the *mpc* of a family of service contracts fulfilling all feature constraints, all necessary service requests and the maximal number of permitted service requests. Building on an earlier model [8], permitted and necessary transitions are interpreted as controllable and uncontrollable transitions in SCT [16]. In this paper, we partitioned necessary transitions based on their degree of controllability, thus enriching the synthesis algorithm by considering necessary transitions whose controllability can be altered due to non-local information.

SCT was previously applied to Software Product Line Engineering in [14], where the `CIF 3` toolset was used to synthesise all valid products of a product line composed of behavioural components and requirements modelled as automata. Based on the synthesis of the *mpc* in [8], our approach to synthesise a family of services does not consider all actions to be controllable, as in [14], but considers increasing levels of uncontrollability (from urgent to lazy requests). The information related to the specific requirements of each product (required and forbidden features) is also integrated into the synthesis algorithm.

Moreover, whilst the number of products is in general exponential in the number of features, the organisation of the family's products (and their *mpc*) into a partial order makes our approach more scalable. As a result, the obtained *mpc* of the family of services can be synthesised from only a subset of its products, whereas other approaches require to synthesise the *mpc* of each single product.

Software architectural and adaptation patterns for DSPL are studied in [21], which are triggered automatically by monitoring executions. An orchestration is assumed to be derived automatically, whilst we focus on how such an orchestration can be synthesised. Moreover, services are described by means of interfaces (i.e. connectors) and are in correspondence with features, whilst we provide their behavioural representation and we identify features as service actions, to be activated according to specific products. FMCA could be used as the underlying formalism for deploying and updating applications, as well as automatically synthesising a well-behaving orchestration of services.

Finally, the presented theory has been implemented in a prototypical tool, which was used to compute all examples given throughout the paper. It remains to compare our approach with other synthesis algorithms and to quantify how well it scales. To this aim, we would like to model and analyse a real world service-based application, as was done in [4] for a system without variability.

Another direction for future work is to enhance service requests and offers with quantities. In Sect. 2, e.g., Clients could express the actual amount of money they are willing to pay. Reaching an agreement would then amount to finding the optimal trade-off among principals such that each one has a positive pay-off function. This might lead to a formalisation of Quality of Service parameters of Service Level Agreements in our model, allowing us to assess non-functional parameters like reliability or energy consumption in a composition of service contracts.

Acknowledgements. This work has been partially supported by the Tuscany Region project POR FESR 2014-2020 SISTER.

REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and J. Montagnat. 2008. Imaging Services on the Grid as a Product Line: Requirements and Architecture. In *Proceedings SOAPL.*

[2] A. Antonik, M. Huth, K.G. Larsen, U. Nyman, and A. Wąsowski. 2008. 20 Years of Modal and Mixed Specifications. *B. EATCS* 95 (2008), 94–129.

[3] M. Bartoletti, T. Cimoli, and R. Zunino. 2015. Compliance in Behavioural Contracts: A Brief Survey. In *Programming Languages with Applications to Biology and Security (LNCS)*, Vol. 9465. Springer, 103–121.

[4] D. Basile, S. Chiaradonna, F. Di Giandomenico, and S. Gnesi. 2016. A stochastic model-based approach to analyse reliable energy-saving rail road switch heating systems. *J. Rail Transp. Plann. Man.* 6, 2 (2016), 163–181.

[5] D. Basile, P. Degano, and G.L. Ferrari. 2016. Automata for Specifying and Orchestrating Service Contracts. *Log. Meth. Comput. Sci.* 12, 4:6 (2016), 1–51.

[6] D. Basile, P. Degano, G.L. Ferrari, and E. Tuosto. 2016. Relating two automata-based models of orchestration and choreography. *J. Log. Algebr. Meth. Program.* 85, 3 (2016), 425–446.

[7] D. Basile, F. Di Giandomenico, and S. Gnesi. 2017. FMCAT: Supporting Dynamic Service Product Lines. In *Proceedings SPLC*, Vol. 2. ACM.

[8] D. Basile, F. Di Giandomenico, S. Gnesi, P. Degano, and G.L. Ferrari. 2017. Specifying Variability in Service Contracts. In *Proceedings VaMoS.* ACM, 20–27.

[9] D. Basile, M.H. ter Beek, F. Di Giandomenico, S. Gnesi, P. Degano, G.L. Ferrari, and A. Legay. 2017. *Controller Synthesis of Contract-based Service Product Lines: Extended Version.* Technical Report 2017-TR-003. ISTI–CNR. http://puma.isti.cnr.it/rmydownload.php?filename=cnr.isti/cnr.isti/2017-TR-003/2017-TR-003.pdf

[10] D.S. Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proceedings SPLC (LNCS)*, Vol. 3714. Springer, 7–20.

[11] M.H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. 2016. Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. *J. Log. Algebr. Meth. Program.* 85 (2016), 287–315.

[12] M.H. ter Beek, S. Gnesi, and M.N. Njima. 2011. Product Lines for Service Oriented Applications - PL for SOA. In *Proceedings WWV (EPTCS)*, Vol. 61. 34–48.

[13] M.H. ter Beek and F. Mazzanti. 2014. VMC: Recent Advances and Challenges Ahead. In *Proceedings SPLC*, Vol. 2. ACM, 70–77.

[14] M.H. ter Beek, M.A. Reniers, and E.P. de Vink. 2016. Supervisory Controller Synthesis for Product Lines Using CIF 3. In *Proceedings ISoLA (LNCS)*, Vol. 9952. Springer, 856–873.

[15] D. Benavides, S. Segura, and A. Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: a Literature Review. *Inf. Syst.* 35, 6 (2010), 615–636.

[16] C.G. Cassandras and S. Lafortune. 2006. *Introduction to Discrete Event Systems.* Springer, New York.

[17] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Trans. Softw. Eng.* 39, 8 (2013), 1069–1089.

[18] G. Cledou and L.S. Barbosa. 2017. Modeling Families of Public Licensing Services: A Case Study. In *Proceedings FormaliSE.* ACM, 37–43.

[19] D. Fischbein, S. Uchitel, and V.A. Braberman. 2006. A foundation for behavioural conformance in software product line architectures. In *Proceedings ROSATEA.* ACM, 39–48.

[20] D. Georgakopoulos and M.P. Papazoglou (Eds.). 2008. *Service-oriented Computing.* MIT Press, Cambridge, MA, USA.

[21] H. Gomaa and K. Hashimoto. 2011. Dynamic Software Adaptation for Service-Oriented Product Lines. In *Proceedings SPLC*, Vol. 2. ACM, 35:1–35:8.

[22] S. Gunther and T. Berger. 2008. Service-Oriented Product Lines: A Development Process and Feature Management Model for Web Services. In *Proceedings SOAPL.*

[23] K. Larsen, U. Nyman, and A.Wąsowski.2007.Modal I/OAutomata for Interface and Product Line Theories. In *Proceedings ESOP (LNCS)*, Vol. 4421. Springer, 64–79.

[24] K. Lauenroth, K. Pohl, and S. Töhning. 2009. Model Checking of Domain Artifacts in Product Line Engineering. In *Proceedings ASE.* IEEE, 269–280.

[25] M. Mannion. 2002. Using First-Order Logic for Product Line Model Validation. In *Proceedings SPLC (LNCS)*, Vol. 2379. Springer, 176–187.

[26] F.M. Medeiros, E.S. de Almeida, and S.R. de Lemos Meira. 2009. Towards an Approach for Service-Oriented Product Line Architectures. In *Proceedings SOAPL.*

[27] M. Raatikainen, V. Myllärniemi, and T. Männistö. 2007. Comparison of Service and Software Product Family Modeling. In *Proceedings SOAPL.*

[28] P.J. Ramadge and W.M. Wonham. 1987. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.* 25, 1 (1987), 206–230.