FMCAT: Supporting Dynamic Service-based Product Lines

Davide Basile
University of Florence, Italy
ISTI–CNR, Pisa, Italy

Felicita Di Giandomenico
ISTI–CNR, Pisa, Italy

Stefania Gnesi
ISTI–CNR, Pisa, Italy

ABSTRACT

We describe FMCAT, a toolkit for Featured Modal Contract Automata (FMCA). FMCAT supports the analysis of dynamic service product lines, i.e., applications consisting of ensembles of interacting services organized as product lines. Services are modelled as FMCA, with features identifying obligations and requirements of services. Service requirements can be either permitted or necessary, whereas the latter are further partitioned according to their criticality. A notion of agreement among service contracts is used to characterise safety.

We show how FMCAT can be used to (*i*) specify dynamic service product line, (*ii*) efficiently identify all valid products, and to synthesise a safe orchestration of services for either (*iii*) a single product, or (*iv*) the whole service product line. FMCAT exploits the theory of FMCA to efficiently perform the above tasks by only visiting a subset of valid products, and it is equipped with a GUI.

CCS CONCEPTS

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

KEYWORDS

Services, Product line, Featured Modal Contract Automata Tool

ACM Reference format:
Davide Basile, Felicita Di Giandomenico, and Stefania Gnesi. 2017. FMCAT: Supporting Dynamic Service-based Product Lines. In *Proceedings of SPLC '17, Sevilla, Spain, September 25-29, 2017,* 6 pages.
DOI: 10.1145/3109729.3109760

1 INTRODUCTION

Service-oriented computing (SOC) [15] is a paradigm for distributed applications based on the publication, discovery and orchestration of *services*. Services are composed to provide Web applications and can be reused in different configurations over time.

Through SOC it is possible to build dynamic service-based applications capable of adapting to changes in the environment or to the resources of the devices on which they run. Services are usually programmed with little or no knowledge about clients and other services before being loosely coupled into networks of collaborating end-user applications. Therefore the idea to organise them into *dynamic service product lines* has been explored at different SPLC conferences (cf., e.g., [18]), leading to applications for Web stores, smart grids and services as used in scientific workflows and grid computing [1, 3, 11] and interest has been recently revived [16]. Concerning SOC, *service contracts* [4] 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 and to services that are no longer available.

Featured modal contract automata (FMCA) have been introduced in [9] for modelling contract-based dynamic service product lines, and are an extension of modal service contract automata [8] and contract automata [5]. An FMCA can model either single services (called *principals*) or compositions of services based on an orchestrated coordination [7]. 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 possible to characterise the behaviour of an ensemble of services. An execution is considered safe if all requests are matched by corresponding offers. Variability mechanisms are available to distinguish *necessary* ($\Box$) from *permitted* ($\Diamond$) requests. Offers are only permitted as dictated by agreement. Necessary service requests can be urgent, greedy or lazy and have, in decreasing order of relevance, further restrictions on their satisfiability.

Features are identified as service actions, and each FMCA represents a behavioural product line of services equipped with feature constraints. Feature models are described as usual, where each product of the product line is identified as a truth assignment satisfying the corresponding feature constraints. Contract agreement guarantees the fulfilment of all feature constraints, all variants of necessary requests and the maximum number of permitted requests that could be fulfilled without spoiling the service composition. Contracts adapt to the overall agreement by renouncing to unsatisfiable, yet permitted requirements.

In this paper we present FMCAT: a prototypical tool implementing the theory of FMCA [9]. FMCAT organises the products into a partial order, to efficiently compute all valid products and the orchestration of the service product line from only a subset of products. FMCAT also allows to compute the orchestration of a single product and features both a GUI and a command-line prompt. FMCAT is open-source and a demo is available at https://github.com/davidebasile/FMCAT.

2 FMCAT FUNCTIONALITIES

We consider a simple franchise of Hotel reservation systems and model it as a service product line. Such a system consists of clients (either business or economy) 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

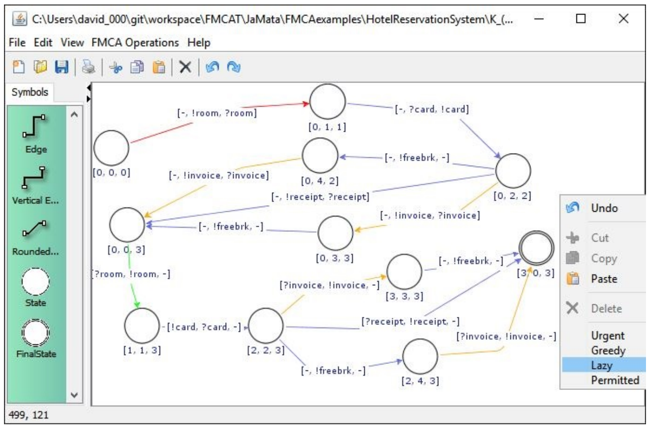


Figure 1: FMCAT at work

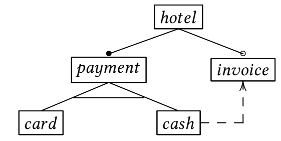


Figure 2: A feature model

Fig. 2 (cf. [9] for more details on this example). The *orchestration* for the composition `EconomyClient⊗Hotel⊗BusinessClient` computed with FMCAT is depicted in Figure 1, specifically for the product requiring features *invoice* and *card* while forbidding feature *cash* (cf. product `p2` below), i.e. payments can only be made through credit card and invoices are required. This orchestration dictates how the computation evolves: at each transition the states of principals and their actions are identified, e.g. `BusinessClient` (third service) is served before `EconomyClient` (first service), and according to the product no cash payment is performed.

Urgent, greedy, lazy and permitted transitions are specified by the colour of the corresponding transitions, which are red, orange, green and blue, respectively. Offer actions are prefixed by ! while requests are prefixed by ?. In our example, the room request (`?room`) of `BusinessClient` is urgent while the same request for `EconomyClient` is lazy, accordingly in their orchestration the urgent request is served before the lazy one.

FMCAT exploits FMCA [9] (see Definition 5.4), and in particular it uses results from Supervisory Control Theory [19] to build a safe orchestration of services as the *most permissive controller* (mpc) of the composition of FMCA. Permitted and necessary actions are interpreted as controllable and uncontrollable actions, respectively. Controllable actions can be blocked by the mpc, while this is not possible for the uncontrollable ones. Indeed, necessary requirements of all service contracts must be fulfilled for obtaining a safe orchestration of services (see Section 6). The main functionalities provided by FMCAT are listed below:

Import/Export FMCA FMCAT features both a command-line interface and a GUI. Accordingly, an FMCA can be specified either through a text file (extension `*.data`) or through the GUI (extension `*.mxe`). It is possible to import textual descriptions of FMCA directly into the XML format for the GUI and vice-versa. In the actual version of the tool, the graphical arrangement of an FMCA (e.g. Figure 1) is not exported to the textual description, where only the information about states and transitions is kept.

Contract Composition the operation of contract composition is an adaptation of the one available for contract automata [6] (see Definition 5.6). FMCA are composable, and an automaton can specify either a single service or a service composition. Indeed, through FMCA it is possible to specify dynamic service product line, where new services can be added to the whole composition at binding time. The operator of composition basically interleaves all the actions of principals, with the only restriction being the case in which two principals are ready on their corresponding request/offer action: in this case only their synchronization (called match) will be available. The FMCA in Figure 1 is a composition of three principals.

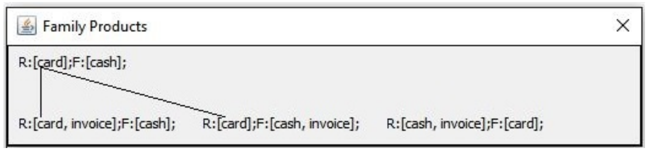
Generation of Partial Order of Products An FMCA consists of a behavioural description of a service (i.e. the automaton) together with a feature model describing the product line. In particular, in FMCA a text file (extension .prod) specifies each product through its set of *required* and *forbidden* actions that are, respectively, true and false atoms of the formula (feature constraints) representing the feature model (see Definition 5.1). An example of family description is below, corresponding to the feature model in Figure 2, also identified by the formula $\varphi = ((card \wedge \neg cash) \vee (cash \wedge \neg card)) \wedge (\neg cash \vee invoice)$. Three products satisfying φ are listed below:

```
p2: R={card,invoice} F={cash}; p3: R={card}
F={cash,invoice}; p4: R={cash,invoice} F={card}
```

Note that in FMCA the leaves of the corresponding feature model are features, and are a subset of service actions. Moreover, FMCAT considers products where not all variability has been resolved (aka sub-families), but sufficient to decide whether the formula is satisfied or not. In this case, the interpretation function (i.e. product) `card`= *true*, `cash`= *false* satisfies φ and has to solve the variability related to the `invoice` feature: this is the product `p1`: `R={card} F={cash}`, identified as a *super*-product of both `p2` and `p3`. Indeed, required and forbidden actions of `p1` are included in its sub-products; and the two "top" products are `p1` and `p4` (see Definition 5.2). FMCAT exploits this ordering relation (i.e. set inclusion) among products to efficiently verify the service product line. In Figure 3 the partial order of the above products is depicted, and it is automatically generated by FMCAT.

Verification of Valid Products Once an FMCA $\mathcal{A}$ has been loaded or imported, together with its partial order of products, all products that are valid in the FMCA $\mathcal{A}$ can identified, i.e. those where all required features (i.e. actions) are available in $\mathcal{A}$, and none of the forbidden features is (see Definition 5.7). By relying on the theory of FMCA, it is possible to identify all such products, potentially exponential in number, without performing the check for each one of them. Indeed, FMCAT internally represents the products through a tree data-structure. The algorithm for this operation basically performs a top-down breadth-first visit of the tree, where sub-trees rooted in products non-valid in $\mathcal{A}$ are pruned. It is known that validity of a product of an FMCA implies validity of all its super-products [9]. Given the FMCA depicted in Figure 1 and the products in Figure 3, the products valid in the FMCA are `p1` and `p2` only.

Computing Canonical Products Canonical products are those



Family Products
R:[card];F:[cash];
R:[card, invoice];F:[cash];
R:[card];F:[cash, invoice];
R:[cash, invoice];F:[card];

Figure 3: A partial order of products generated by FMCAT

characterising the whole service product line. All other services of the given family can be obtained by refinement of some canonical product. Canonical products are all valid "top" products of the given partial order, quotiented by their set of forbidden features. FMCAT allows to identify all canonical products from an FMCA and its products. In the above example, p1 is the only canonical product of the service product line. Indeed, the orchestration of p2 is contained into the one of p1.

Orchestration of a Product An orchestration of services is the maximal sub-portion of the FMCA that is *safe*, i.e. all requests of services are matched by corresponding offers (e.g. the FMCA in Figure 1 is safe). The orchestration, being the mpc from Supervisory Control Theory, tries to keep the maximum number of permitted actions, while necessary requests must be matched for reaching a non-empty orchestration. Urgent, greedy and lazy necessary requests characterise "when" the request can be matched, and give rise to different priorities. Urgent requests do not allow delays (due to interleavings generated by the composition), i.e. they are uncontrollable. For example, the red transition in Figure 1 is matched in the initial state. Greedy (orange) requests can be delayed as soon as the first match is available, that is, greedy matches are uncontrollable. Lazy matches/requests can be controlled by the orchestration, provided that at least one match is available. For example, in Figure 1, the green request of the first principal (EconomyClient) is served after the red request of the third principal (BusinessClient).

FMCAT computes an orchestration for a single product, where all its required actions must be available in the orchestration, and none of the forbidden actions is available.

Orchestration of a Service Product Line Through FMCAT it is also possible to compute the orchestration of a whole service product line. By exploiting theoretical results from [9], the orchestration can be computed without iterating through each product. In particular, the orchestration of the service product line is the union of the orchestrations of all canonical products. The FMCA in Figure 1 is the orchestration of the canonical product p1 and hence it is also the orchestration of the whole service product line, identified by the feature model in Figure 2. Indeed, the orchestration of p2, in this case, is exactly the one of p1 (in general it could be a sub-automaton). If, for example, we would add a fifth product p5: R={receipt,invoice}, F={taxi} (obtained by modifying the feature model), then this would be another canonical product and the union of the orchestrations of p1 and p5 would be the mpc of the service product line.

3 FMCAT ARCHITECTURE

FMCAT is based on a previous tool for contract automata (CA) [6], implemented in Java. In particular, FMCAT extends the main classes used for CA by adding the new functionalities described in Section 2. The Contract Automata Tool also relies on a previous framework

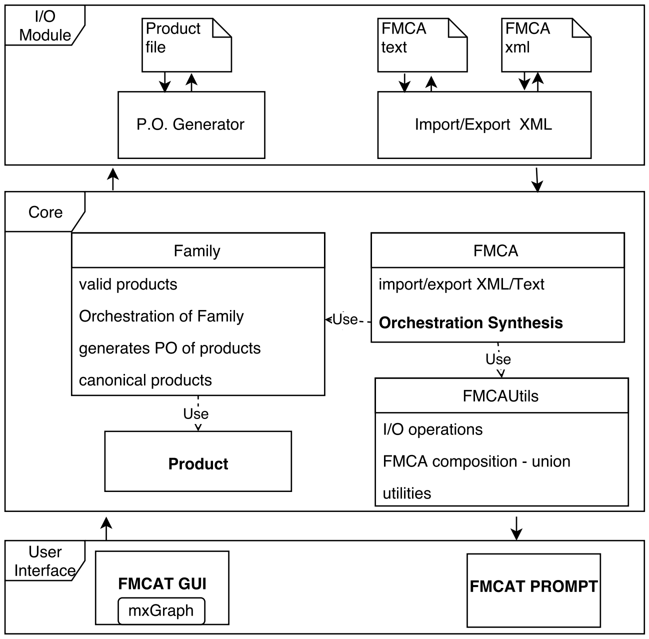


Figure 4: The Architecture of FMCAT

for specifying Finite State Automata in Java, from which it inherited some basic functionalities for storing and printing automata.

A Diagram showing the architecture of FMCAT is depicted in Figure 4, and it is composed of three main modules:

User Interface This module contains the two Java classes FMCATGUI.java and FMCATPROMPT.java. The first one implements the GUI of FMCAT, and it is based on an existing framework called *mxGraph* for editing graphs in Java. The command line prompt is mainly inherited from the previous versions of the tool and it allows to interact with the API of FMCAT by means of an interactive command-line interface.

I/O Module This module concerns with the storing of file descriptors used by FMCA. There are three types of files used by the tool: *.prod files contain textual descriptions of products as described previously. An FMCAT is stored in either a readable textual representation (*.data) or an XML format (*.mxe). The *.data format is mainly used by the command-line interface and it consists of a textual declaration of states and transitions of each automaton. The XML representation of FMCA (*.mxe) is used by the GUI for saving and loading FMCA. This file descriptor also stores information related to the graphical visualization of the FMCA. The module also contains the Partial Order Generation from a *.prod file, computed when the product files are loaded, and it contains the utilities for converting a description of FMCA into one of the available formats.

Core The core module of FMCAT is composed of, among the others, the class FMCA.java, extending CA.java and implementing the algorithm for synthesising a safe orchestration of services for a given product. This class uses two others: FMCAUtils.java is a stand-alone class offering functionalities for composing automata, for computing the union of FMCA (used for synthesising the orchestration of a family) and other utilities. The class Family.java uses another class `Product.java` for memorising the various products composing a given product line. It contains the methods for computing the valid products of a family, for computing the partial order of products and the canonical product. It is also used by the `FMCA` class for computing the orchestration of the service product line (i.e. the union of the mpc of the canonical products).

4 RELATED WORK AND CONCLUSION

In this paper, we discussed FMCAT, a tool supporting dynamic service-based product lines, based on featured modal contract automata. FMCAT features a GUI and a command-line interface, and organises the product line into a partial order for efficiently compute a safe orchestration of services, specifying a dynamic service-based product line application.

FMCAT is an extension of a previous tool called Contract Automata Tool (CAT) [6]. Whilst FMCAT is tailored to Service Product Line, CAT tackled the problem of verifying circularity issues among service contracts, modelled as flow problem and solved through linear integer programming techniques. CAT also tackled the problem of translating an orchestration of services into a choreography. It was based on a command line interface, while FMCAT also offers a user-friendly GUI. Below we discuss some tools that can be used for editing and managing feature models, to be solved and imported into the FMCAT format (`*.prod`).

FeatureIDE [22] is an open-source framework for feature-oriented software development based on Eclipse. It uses different feature models management tools and has a Java API to manipulate feature models. A tool for feature models edits [21] has been integrated into FeatureIDE. FAMILIAR [2] is a Domain-Specific Language (DSL) that is dedicated to the large scale management of feature models. It provides operations for decomposing, aggregating and merging several feature models. It can be coupled with FeatureIDE to reason about feature models.

A compositional modelling framework for dynamic product lines is discussed in [14], that relies on annotated versions of probabilistic automata with costs. Compared to [14], composition of FMCA can vary also depending on the order of the operands and different critical levels of actions are available, thus adding an extra layer of expressiveness, whilst in [14] a standard synchronous composition with messages broadcast is available. Moreover, FMCAT avoids to enumerate all products of a product line, whilst [14] may not.

Supervisory Control Theory was previously applied to Software Product Line Engineering in [12], 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. Our approach to synthesise a family of services does not consider all actions to be controllable, as in [12], 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, the organisation of the family's products (and their mpc) into a partial order makes our approach more scalable. As a result, in FMCAT 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.

The FMCA Tool is still under development, and the currently available implementation has been used by the authors for developing the theory of FMCA and exploring the problem of specifying service-based dynamic service product lines. Possible future extensions of FMCAT are listed below. Some functionalities of CAT (i.e. choreography, circularity) have not been included in FMCAT, and their feasibility in a product line environment requires more research from a theoretical viewpoint. A utility for importing directly into the FMCAT format different feature models computable through other tools (e.g. FeatureIDE) is also under development. A static check of the user input is also a future improvement, to help users in specifying FMCA correctly through the GUI. We are also planning to compare our tool with others existing in the literature to emphasise pros and cons of our approach.

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

REFERENCES

[1] M. Acher, P. Collet, A. Gaignard, P. Lahire, J. Montagnat, and R.B. France. 2012. Composing multiple variability artifacts to assemble coherent workflows. *Softw. Qual. J.* 20, 3 (2012), 689–734.

[2] M. Acher, P. Collet, P. Lahire, and R.B. France. 2013. FAMILIAR: A domain-specific language for large scale management of feature models. *Science of Computer Programming* 78, 6 (2013), 657 – 681.

[3] 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.*

[4] 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.

[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. Playing with Our CAT and Communication-Centric Applications. In *FORTE 2016*. Springer, 62–73.

[7] 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.

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

[12] 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.

[13] 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.

[14] C. Dubslaff, C. Baier, and S. Klüppelholz. 2015. *Probabilistic Model Checking for Feature-Oriented Systems.* Springer Berlin Heidelberg.

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

[16] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and P. Heymans. 2017. Yo Variability! JHipster: A Playground for Web-apps Analyses. In *Proceedings VaMoS*. ACM, 44–51.

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

[18] 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.*

[19] 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.

[20] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. 2006. Feature Diagrams: A Survey and a Formal Semantics. In *Proceedings RE*. IEEE, 136–145.

[21] T. Thum, D. Batory, and C. Kastner. 2009. Reasoning About Edits to Feature Models. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, 254–264.

[22] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. 2014. FeatureIDE: An Extensible Framework for Feature-oriented Software Development. *Sci. Comput. Program.* 79 (2014), 70–85.

5 APPENDIX

In the following FMCA are recalled. See [9] for more details.

5.1 Feature Models

A feature model is a rooted and/or tree in which nodes are features and additional relations between nodes model further constraints (typically mandatory, optional or alternative, but also requires and xor) [13, 20]. It is well known that a feature model is equivalent to a propositional formula over features. Thus, checking the validity of a product with respect to the feature model reduces to a Boolean satisfiability problem, efficiently computable with BDD or SAT solvers [10, 17]. Following [10, 20], we distinguish compound features (intermediate, decomposable nodes) and primitive features (influencing final products). The latter are represented by the leaves of a feature model and the propositional formula representing a feature model uses only them as literals.

In our framework, we distinguish basic actions belonging to the sets of *requests* $\mathbb{R} = \{a, b, c, \ldots\}$ and *offers* $\mathbb{O} = \{\overline{a}, \overline{b}, \overline{c}, \ldots\}$ where $\mathbb{R} \cap \mathbb{O} = \emptyset$. Primitive features are identified as basic actions. A *feature constraint* is a propositional logic formulae φ over $\mathbb{R} \cup \mathbb{O}$.

A service product line is then characterised by a conjunction of feature constraints with literals in $\mathbb{R} \cup \mathbb{O}$, such that each assignment p satisfying φ (written $\varphi \models_p$ *true*) is a *valid product*.

Definition 5.1 (Products). Let φ be a conjunction of feature constraints with literals in $\mathbb{R} \cup \mathbb{O}$ and let $\mathcal{P} : \mathbb{R} \cup \mathbb{O} \Rightarrow \{true, false\}$ be an interpretation function. Then $[\![\varphi]\!] = \{\, p \mid \varphi \models_p true \text{ and } p \in \mathcal{P} \,\}$ is the set of all products of φ. Moreover, given $p \in [\![\varphi]\!]$, the sets of required and forbidden actions in p are $Required(p) = \{\, a \mid p(a) = true \,\}$ and $Forbidden(p) = \{\, a \mid p(a) = false \,\}$, respectively.

All products $[\![\varphi]\!]$ of a family can be ordered by component-wise set inclusion as (a subset of) elements of a lattice such that the bottom element $\perp$ requires and forbids all actions, whereas the top element $\top$ has neither required nor forbidden actions. Note that not all elements of such a lattice correspond to products of $[\![\varphi]\!]$.

Definition 5.2 (Sub-products). Let $(R, F) \subseteq (R', F') \in ((\mathbb{R} \cup \mathbb{O}) \times (\mathbb{R} \cup \mathbb{O}), \subseteq)$ be a lattice iff $R \subseteq R'$ and $F \subseteq F'$. The partial order of products of a family $[\![\varphi]\!]$ is $([\![\varphi]\!], \preceq)$, where $p \preceq p'$ (p is a sub-product of p' or, alternatively, p' is a super-product of p) iff

$$(Required(p'), Forbidden(p')) \subseteq (Required(p), Forbidden(p))$$

5.2 FMCA

We now formally define featured modal contract automata (FMCA), which extend modal service contract automata (MSCA) [8].

We borrow some useful notation from [5, 7]. The alphabet of *basic actions* is defined as $\Sigma = \mathbb{R} \cup \mathbb{O} \cup \{\bullet\}$ where $\bullet \notin \mathbb{R} \cup \mathbb{O}$ is a distinguished element representing the *idle* move. We define the involution $co(\bullet) : \Sigma \mapsto \Sigma$ s.t. $co(\mathbb{R}) = \mathbb{O}$, $co(\mathbb{O}) = \mathbb{R}$ and $co(\bullet) = \bullet$.

Let $\vec{v} = (e_1, ..., e_n)$ be a vector of *rank* $n \geq 1$, denoted by r_v, and let $\vec{v}_{(i)}$ denote the ith element with $1 \leq i \leq r_v$. By $\vec{v}_1\vec{v}_2 \cdots \vec{v}_m$ we denote the concatenation of m vectors $\vec{v}_i$. From now onwards, we stipulate that in an action vector $\vec{a}$ there is either a single offer or a single request, or a single pair of request-offer that matches, i.e. there exists exactly i, j such that $\vec{a}_{(i)}$ is an offer and $\vec{a}_{(j)}$ is the complementary request or vice versa; all the other elements of the vector contain the symbol $\bullet$, meaning that the corresponding principals remain idle. In the following, let $\bullet^m$ denote a vector of rank m, all elements of which are $\bullet$. Formally:

Definition 5.3 (Actions). Given a vector $\vec{a} \in \Sigma^n$, if

- $\vec{a} = \bullet^{n_1}\alpha\bullet^{n_2}, n_1, n_2 \geq 0$, then $\vec{a}$ is a *request (action) on* α if $\alpha \in \mathbb{R}$, whereas $\vec{a}$ is an *offer (action) on* α if $\alpha \in \mathbb{O}$
- $\vec{a} = \bullet^{n_1}\alpha\,\bullet^{n_2}\,co(\alpha)\bullet^{n_3}, n_1, n_2, n_3 \geq 0$, then $\vec{a}$ is a *match (action) on* α, where $\alpha \in \mathbb{R} \cup \mathbb{O}$

Actions $\vec{a}$ and $\vec{b}$ are *complementary*, denoted by $\vec{a} \bowtie \vec{b}$, if and only if the following holds: (i) $\exists \alpha \in \mathbb{R} \cup \mathbb{O}$ s.t. $\vec{a}$ is either a request or an offer on α; (ii) $\vec{a}$ is an offer on α implies that $\vec{b}$ is a request on $co(\alpha)$; (iii) $\vec{a}$ is a request on α implies that $\vec{b}$ is an offer on $co(\alpha)$.

The actions and states of contract automata are vectors of basic actions and states of principals, respectively. The alphabet of an FMCA consists of vectors, each element of which intuitively records the execution of basic actions of principals in the contract.

An FMCA declares a contract-based service product line through (i) *permitted* and *necessary* transitions; and (ii) a conjunction of feature constraints φ identifying all valid products. We recall that all offers are permitted. Permitted offers and requests are optional and can be discarded.

The set of necessary requests of an FMCA is further partitioned into *urgent*, *greedy* and *lazy*. These sets contain *necessary* requests that must be matched to reach an agreement among contracts. We thus offer modellers another layer of variability based on the possibility to specify "when" requests must be matched in a composition.

Definition 5.4 (Featured modal contract automata). Assume as given a finite set of states $\mathfrak{Q} = \{q_1, q_2, \ldots\}$. Then a *featured modal contract automaton* $\mathcal{A}$, FMCA for short, of rank $n \geq 1$ is a tuple $\langle Q, \vec{q_0}, A^{\Diamond}, A^{\Box_u}, A^{\Box_g}, A^{\Box_\ell}, A^o, T, \varphi, F \rangle$, where

- $Q = Q_1 \times \cdots \times Q_n \subseteq \mathfrak{Q}^n$
- $\vec{q_0} \in Q$ is the initial state
- $A^{\Diamond}, A^{\Box_u}, A^{\Box_g}, A^{\Box_\ell} \subseteq \mathbb{R}$ are (pairwise disjoint) sets of permitted, urgent, greedy and lazy requests, resp., and we denote by $A^r = A^{\Diamond} \cup A^{\Box_u} \cup A^{\Box_g} \cup A^{\Box_\ell}$ the set of requests
- $A^o \subseteq \mathbb{O}$ is the finite set of offers
- $T \subseteq Q \times A \times Q$, where $A = (A^r \cup A^o \cup \{\bullet\})^n$, is the set of transitions partitioned into *permitted* transitions $T^{\Diamond}$ and *necessary* transitions $T^{\Box}$ with $T = T^{\Diamond} \cup T^{\Box}$ such that, given $t = (\vec{q}, \vec{a}, \vec{q'}) \in T$, the following holds:
  - $\vec{a}$ is either a request or an offer or a match
  - $\forall i \in 1 \ldots n,\ \vec{a}_{(i)} = \bullet$ implies $\vec{q}_{(i)} = \vec{q}'_{(i)}$
  - $t \in T^{\Diamond}$ iff $\vec{a}$ is either a request on $a \in A^{\Diamond}$, an offer on $\overline{a} \in A^o$ or a match on $a \in A^{\Diamond} \cup A^o$
  - $t \in T^{\Box}$ iff $\vec{a}$ is either a request $a \in A^{\Box_u} \cup A^{\Box_g} \cup A^{\Box_\ell}$ or a match on $a \in A^{\Box_u} \cup A^{\Box_g} \cup A^{\Box_\ell} \cup A^o$
- φ is a conjunction of feature constraints
- $F \subseteq Q$ is the set of final states

A *principal* FMCA (or just *principal*) has rank 1 and $A^r \cap co(A^o) = \emptyset$.

5.3 Composing FMCA

The FMCA operators of composition are crucial for specifying dynamic service product lines, 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 and to synthesise, if possible, a composition satisfying all requirements defined by the service contracts.

A set of FMCA is *composable* if and only if the conjunction of their feature constraints leads to no contradiction.

Definition 5.5 (Composable). A set $Set = \{\mathcal{A}_i \mid i \in 1 \dots n\}$ of FMCA is *composable* iff $(\bigwedge_{\mathcal{A}_i \in Set} \varphi_{\mathcal{A}_i}) \not\models \mathit{false}$.

The operands of the composition $\otimes$ are either principals or composite services. Intuitively, the product composition interleaves the actions of all operands, with the only restriction that if two operands are ready to execute two complementary actions ($\vec{a}_i \bowtie \vec{a}_j$) then only their match will be allowed and their interleaving prevented. Below we use $\bigcirc$ as a placeholder for both necessary ($\Box$) and permitted ($\Diamond$) transitions. More in detail, the transitions of the composite service are generated as follows. Case (1) in Definition 5.6 generates match transitions starting from two operands' transitions having complementary actions ($\vec{a}_i \bowtie \vec{a}_j$). If, e.g., $(\vec{q}_j, \vec{a}_j, \vec{q}_j') \in T^{\Box}$, then the resulting match transition will be marked as necessary (i.e. $(\vec{q}, \vec{c}, \vec{q}') \in T^{\Box}$). If both operands' complementary actions are permitted, then their resulting match transition t will be marked as permitted. All other principals not involved in t will remain idle.

Case (2) in Definition 5.6 generates all interleaved transitions only if no complementary actions can be executed from the composed source state (i.e. $\vec{q}$). In this case, an operand executes its transition $t = (\vec{q}_i, \vec{a}_i, \vec{q}_i')$ and all other operands remain idle. The composed transition will be marked as necessary (permitted) only if t is necessary (permitted, respectively). Note that condition $\vec{a}_i \bowtie \vec{a}_j$ excludes pre-existing match transitions of the operands from generating new matches. Recall that we implicitly assume the set of labels of an FMCA of rank m to be $A \subseteq (A^r \cup A^o \cup \{\bullet\})^m$.

Definition 5.6 (Composition). Let $\mathcal{A}_i$ be composable FMCA of rank r_i, $i \in 1, \dots, n$, and let $\bigcirc \in \{\Diamond, \Box\}$. The *product composition* $\bigotimes_{i \in 1 \dots n} \mathcal{A}_i$ is the FMCA $\mathcal{A}$ of rank $m = \sum_{i \in 1 \dots n} r_i$, where

- $Q = Q_1 \times \cdots \times Q_n$, with $\vec{q_0} = \vec{q_0}_1 \cdots \vec{q_0}_n$
- $A^r = \bigcup_{i \in 1 \cdots n} A_i^r$, $A^o = \bigcup_{i \in 1 \cdots n} A_i^o$,
- $T^{\bigcirc} \subseteq Q \times A \times Q$ s.t. $(\vec{q}, \vec{c}, \vec{q}') \in T^{\bigcirc}$ iff, when $\vec{q} = \vec{q}_1 \cdots \vec{q}_n \in Q$,
  (1) either there are $1 \leq i < j \leq n$ s.t. $(\vec{q}_i, \vec{a}_i, \vec{q}_i') \in T_i^{\bigcirc}$, $(\vec{q}_j, \vec{a}_j, \vec{q}_j') \in T_j^{\bigcirc \cup \Diamond}$, $\vec{a}_i \bowtie \vec{a}_j$ and
  $$\begin{cases} \vec{c} = \bullet^u \vec{a}_i \bullet^v \vec{a}_j \bullet^z, \text{with } u = r_1 + \cdots + r_{i-1}, \\ v = r_{i+1} + \cdots + r_{j-1},\ z = r_{j+1} + \cdots + r_n,\ |\vec{c}| = m \\ \text{and } \vec{q}' = \vec{q}_1 \cdots \vec{q}_{i-1}\, \vec{q}_i'\, \vec{q}_{i+1} \cdots \vec{q}_{j-1}\, \vec{q}_j'\, \vec{q}_{j+1} \cdots \vec{q}_n \end{cases}$$
  (2) or there is $1 \leq i \leq n$ s.t. $(\vec{q}_i, \vec{a}_i, \vec{q}_i') \in T_i^{\bigcirc}$ and
  $$\begin{cases} \vec{c} = \bullet^u \vec{a}_i \bullet^v, \text{with } u = r_1 + \cdots + r_{i-1}, \\ v = r_{i+1} + \cdots + r_n, |\vec{c}| = m, \\ \vec{q}' = \vec{q}_1 \cdots \vec{q}_{i-1}\, \vec{q}_i'\, \vec{q}_{i+1} \cdots \vec{q}_n \text{ and } \forall j \neq i, 1 \leq j \leq n \\ \text{s.t. } (\vec{q}_j, \vec{a}_j, \vec{q}_j') \in T_j^{\bigcirc \cup \Diamond},\ \vec{a}_i \bowtie \vec{a}_j \text{ does not hold} \end{cases}$$
- $\varphi = \bigwedge_{i \in 1 \dots n} \varphi_i$
- $F = \{ \vec{q}_1 \cdots \vec{q}_n \mid \vec{q}_1 \cdots \vec{q}_n \in Q,\ \vec{q}_i \in F_i,\ i \in 1 \dots n \}$

5.4 Valid Products of FMCA

Intuitively, a *valid product* p of an FMCA $\mathcal{A}$ is such that all its required actions are available while its forbidden actions are not. More precisely, p is a (valid) interpretation of the feature constraints of $\mathcal{A}$ (i.e. $p \in [\![\varphi_{\mathcal{A}}]\!]$) such that for all *true* literals a (i.e. $a \in Required(p)$) a reachable transition t on a is executable in $\mathcal{A}$, whereas for all *false* literals b (i.e. $b \in Forbidden(p)$) no reachable transition t on b can be executed in $\mathcal{A}$. Let $Dangling(\mathcal{A})$ denotes the unreachable states of $\mathcal{A}$ from initial or final states. Formally:

Definition 5.7 (Valid product). Let $\mathcal{A}$ be an FMCA, then $p \in [\![\varphi_{\mathcal{A}}]\!]$ is valid in $\mathcal{A}$ iff (i) $\forall a \in Required(p)\ \exists(\vec{q}, \vec{a}, \vec{q}') \in T_{\mathcal{A}}$ s.t. $\vec{a}$ is an action on a and $\vec{q}' \notin Dangling(\mathcal{A})$, and (ii) $\forall b \in Forbidden(p)$ $\nexists(\vec{q}, \vec{b}, \vec{q}') \in T_{\mathcal{A}}$ s.t. $\vec{b}$ is an action on b and $\vec{q}' \notin Dangling(\mathcal{A})$.

Given a service product line $\mathcal{A}$, one of the benefits of adopting a partial order of products is the possibility to determine all valid products in $\mathcal{A}$ by only exploring a subset of them, as proved in the following theorem. In particular, if a sub-product p is valid then all its super-products p' are also valid products or, equivalently, if a product p' is not valid then neither is any of its sub-products p.

6 CONTROLLER SYNTHESIS

We describe an algorithm for synthesising an orchestration of FMCA, viz. the maximal sub-portion of an FMCA $\mathcal{A}$ that is safe. The orchestration will be the *most permissive controller* (*mpc* for short) in the style of Supervisory Control for Discrete Event Systems [19].

The purpose of contracts is to declare all executions of a principal in terms of requests and offers. Therefore, all actions of a (composed) contract are observable.

The behaviour to enforce upon a given FMCA are exactly 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, a composition is forced to be in agreement only if there exists a match for each necessary (urgent, greedy or lazy) request.

We now outline the iterative algorithm for computing the *mpc* of product p of an FMCA $\mathcal{A}$. With respect to the standard synthesis in [19], non-local information related to other transitions is exploited 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, all possible (non-matched) requests must be removed, and also 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 the bad uncontrollable transitions can only be made 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 proceeds backwards from final states, to discover new states to be added to R_i. 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.