Managing Trade-offs among Architectural Tactics using Feature Models and Feature-Solution Graphs

Jaime Chavarriaga*†, Carlos Noguera†, Rubby Casallas*, and Viviane Jonckers†
*Universidad de los Andes, Bogota, Colombia
{ja.chavarriaga908, rcasalla}@uniandes.edu.co,
†Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium
{jchavarr, cnoguera, vejoncke}@vub.ac.be

***Abstract*—In Software Architecture, several approaches focus first on selecting architectural tactics to address quality attributes and later use the corresponding design alternatives to create application architectures. Regrettably, sometimes the alternatives used to improve some quality attributes inhibit or reduce the achievement of others. These conflicts, known as trade-off points, lead to trade-off decisions to solve them. Usually, detection of trade-off points and guidance for trade-off decisions rely on the expertise of software architects. The architect has to (1) identify and analyze the conflict on design alternatives, (2) determine which quality attributes and tactics motivated the selection of conflicting alternatives, and (3) decide about which set of non-conflicting tactics to use. This paper presents an approach based on feature models to help manage trade-offs. It is based on the specification of relationships between architectural tactics and design alternatives that describe, for each tactic, which combination of designs can be used or must not be used. When a set of tactics are selected to implement an architecture, these relationships serve to determine the set of alternatives to use. In addition, they aid to detect trade-off points and determine the tactics to consider in a trade-off decision. We present a formalization of our approach and illustrate it using a case study on tactics and patterns for database-based web applications.**

***Index Terms*—Software Product Lines, Quality attributes, Feature Model, Feature-Solution Graphs**

I. Introduction

A *software architecture tactic* is a reusable solution, akin to a design pattern, that describes which design alternatives help to achieve a quality attribute such as performance, availability, maintainability or security [1][2]. Software architects must understand the tactics and select proper design alternatives according to the type of applications or technologies they are using, as well as the quality attributes they stive for.

For instance, a software architect might select the *maintain copies of computations* tactic to improve the performance of an application [1]. This tactic aims to reduce the contention that would occur if all computations took place on a central server. It can be implemented using diverse design alternatives such as *load-balanced web servers*, *clustered application servers*, or *clustered database servers*.

Conflicts may appear when the software architect select additional tactics and design alternatives. A *trade-off* [1][2] occurs when an architect chooses design alternatives that helps to achieve some of the requirements but inhibits or reduce the satisfaction of others. For instance, consider an application striving for both performance and availability. To achieve performance, an architect may select the *maintain copies of computations* tactic and the *load-balanced web servers* design alternative using a fast *TCP/IP load balancer* that does not check or process the headers in the web requests. On the other hand, to achieve availability, the same software architect may select the *Active redundancy* tactic [1] using the *high-availability clusters* alternative. However, clustering for web servers requires an HTTP load balancer that process headers of the web protocol. Without noticing it, the architect has chosen design alternatives that leads to a trade-off.

In a *trade-off*, the *trade-off point* [2] is the set of conflicting design alternatives that affect more than one quality attribute. Usually, these points appear because a tactic *suggests* the use of an alternative while another tactic *prohibits* its use. It is the job of software architects to perform *trade-off decisions* to solve such trade-off points. A trade-off decision involves the selection of a combination of tactics for which the corresponding design alternatives do not have conflicts.

Architectural design methods such as ATAM [3] and CBAM [4] suggest that architects must choose, for any pair of tactics that leads to trade-off points, only the tactic aimed to improve the quality attributes with higher priority. Trade-off decisions have been traditionally explained using a matrix [5][6] describing which pairs of the tactics or quality attributes originally considered result in trade-off points.

Despite the existing approaches, trade-off resolution remains a challenge for software architects. Existing approaches [3][4] focus on walk-through evaluations that rely on the experience of peer architects. Other approaches [7] require the creation of simulation models based on complete architectural designs to detect trade-off points caused by tactics for quality attributes such as performance or availability. None of these approaches supports detection of trade-off points caused by conflicts on design alternatives during selection of tactics, without creating a complete architectural design.

This paper presents FaMoSA (Feature Models for Software Architecture), our approach to support early detection of trade-off points and explanation of trade-off decisions. It is based on a model-based specification of architectural tactics, design alternatives, and the relationships among them. In FaMoSA, feature models [8][9] are used to represent architectural tactics and design alternatives, and feature-solution graphs [10][11] are used to specify, for each tactic, which combination of design alternatives must be used or must be not used in an application. Using a feature-based configuration process [12], a software architect may select a set of tactics and design alternatives that implement to these tactics. If these selections result in conflicting design alternatives, our approach is able to detect trade-off points and assist the architect in formulating a trade-off decision by reporting the conflicting design alternatives and presenting possible solutions.

The major contributions of this paper are: 1) the specification of architectural tactics, design alternatives, and the relationships among them using feature models and feature–solution graphs, 2) the configuration of architectural tactics and the identification of either the corresponding non-conflicting design alternatives or the conflicts that lead to trade-off decisions, 3) the explanation of detected trade-off points in terms of architectural tactics.

The remainder of this paper is structured as follows. Section II introduces a motivating example of architectural tactics and design alternatives in the domain of database-based web applications. Section III describes our proposal to detect trade-off points and explain trade-off decisions based on feature models and relationships between them. In Section IV, we retake the motivating example, illustrating the application of our approach. Finally, Section V gives an overview of related work, and Section VI concludes the paper.

II. Tactics and Design Alternatives for Availability and performance in database-based web applications – An Example

This section presents a motivating example for our approach in the context of database-based web applications. Here we present tactics introduced by Bass et al. [1] for two quality attributes, Availability and Performance, modeling them as a Feature Model. We also present how these tactics can be implemented using design alternatives to implement web applications. An example of a trade-off is also introduced.

A. Architectural tactics

Architectural tactics are reusable across application domains and technologies. For instance, Bass et al. [1] have proposed tactics for *performance* such as :

- *maintain copies of data* to avoid multiple processing of data;
- *maintain copies of computation* to reduce the contention that would occur if all computations took place on a central server,
- *introduce concurrency* to reduce response time by processing data in parallel, and
- *reduce overhead* to improve processing time by using design alternatives that demand few computation resources.

In addition, same authors have proposed tactics for *availability* such as:

- *active redundancy* where redundant components are arranged to respond events in parallel and to assume the tasks of other components after a failure, and
- the use of *transactions* to prevent erroneous modifications of data that can lead to application failures.

B. Architectural tactic feature model

To model the different architectural tactics in our example, we choose an *Architectural tactic feature model*. These models represent a set of quality attributes and their corresponding architectural tactics [13].

Figure 1a shows the architectural tactic feature model representing the tactics above mentioned. In the figure, optional features are denoted by the clear circle above them,e.g., Introduce Concurrency is optional for the tactics related to performance. Note as well that a priori, there is no restriction on which features (tactics) can be selected; an architect can decide to apply both availability tactics.

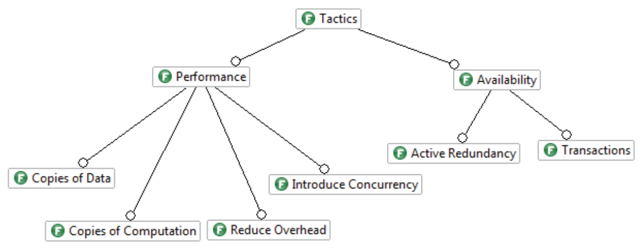
C. Design alternatives for database-based web applications

Design alternatives depend of the type of application and technologies being used. Our running example considers the domain of database-based web applications that run on application servers, processing requests from the user, performing queries over databases and generating responses combining database results with elements in XML or HTML.

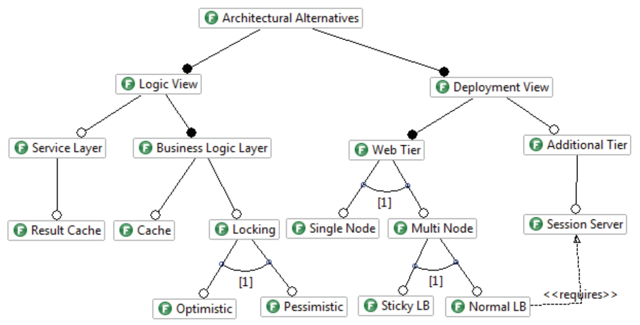
In our example, from the development viewpoint, there are design alternatives for the components that provide services to external applications (e.g., a service layer), and the components that implement the logic of the application (i.e., the logic layer). Architects must also decide on the types of components to include in each layer. For instance, they can use a *result cache* in applications that expose functionality using a service layer (e.g. using SOA or REST protocols) to keep local copies of service responses and avoid the processing of similar requests in the future. In addition, they can use an *entity cache* to keep local copies of data to reduce the number of requests to databases.

In addition, architects may consider different alternatives to manage concurrency in the logic of the application. Opting for *optimistic locking* where users can read stale data and perform updates that fail if someone else has modified the same data before, or *pessimistic locking* where other users cannot try to alter data being reviewed or edited for someone else.

On the other hand, software architects must decide on the deployment of the application. In our example, architects can decide to deploy the application using a *web tier* either on a *single node* or on *multiple nodes*. For the latter, they can use an *sticky-session load balancer* where all the request from the same session (i.e. browser) are redirected to the same node, or a *normal load balancer* where the requests of the same user may be redirected to different nodes each time. The use of multi-node web tiers using normal load balancers requires an additional tier of nodes running a *session server* to maintain session data independently of the node attending the user requests.



(a) Feature model of architectural tactics.



(b) Feature model of architectural design alternatives.

Fig. 1: Feature models representing architectural tactics and design alternatives.

D. *Design alternatives feature model*

Figure 1b shows an architectural feature model representing the design alternatives for *application architecture*, organizing them considering the mentioned *development* and *deployment* architectural viewpoints.

E. *Tactic implementation in database-based web applications*

While architectural tactics are independent of the application domain, design alternatives are not. Thus, the design alternatives used to implement each tactic varies from a type of application to the other.

Table I shows a mapping of the architectural tactics and their corresponding implementing design alternative. The term *forces* indicates that a tactic requires always a specific design alternative, *suggests* indicates that the design alternative is required sometimes, and *prohibits* indicates that a design alternative should not be used.

TABLE I: Example implementation of architectural tactics

Architectural Tactic	**Design Alternatives**
Maintain copies of data	suggests *result cache* forces *entity cache*
Maintain copies of computation	forces *multi-node* web tier
Introduce concurrency	forces *optimistic locking*
Reduce overhead	prohibits *pessimistic locking* suggests *sticky-session load balancer*
Active redundancy	forces *normal load balancer*
Transactions	prohibits *optimistic locking* prohibits *session server*

F. *Design alternatives for a set of tactics*

So far, we have described a set of tactics and their implementation using design alternatives for database-based web applications. For a given tactic, the identification of the corresponding design alternatives is straightforward using the information in Table I. For instance, according to that table the alternatives to implement the *introduce concurrency* tactic for performance and the *active redundancy* for availability are *optimistic locking*, and *normal load balancers*.

G. *Trade-offs between conflicting tactics*

Trade-offs occurs when the tactics selected by an architect have conflicts between them. In our example, if an architect selects *introduce concurrency* for performance and *transactions* for availability, the corresponding design alternatives have a conflict because the first is implemented using *optimistic locking* and the second is implemented by not using it.

A more complex conflict arises if an architect selects *reduce overhead* tactics for performance and *active redundancy* for availability. The reduce overhead tactic suggests sticky-session load balancing, while active redundancy forces a normal load balancing feature. These two types of load balancing are conflicting, since the design alternatives feature model constraints them to be exclusive (only one of them can be active); thus generating a trade-off point.

As the number of design alternatives, dependencies, and constraints of an application increases, the detection of trade-off points becomes less trivial.

III. Detecting Trade-off Points and Explaining Trade-off Decisions

This section presents FaMoSA (Feature Models for Software Architecture), our approach to detect trade-off points and explain trade-off decisions based on the feature model-based specification of architectural tactics, design alternatives, and the relationships among them. We give first a general overview and then a formal specification of our approach.

A. *Approach overview*

FaMoSA's strategy is threefold:

1) *the specification of architectural tactics and design alternatives* using feature models, as well as and the specification of the relationships between them as feature solution graphs (FS-Graphs) (section III-B),
2) *the configuration of architectural tactics* and the identification of either the corresponding non-conflicting design alternatives or the conflicts that lead to a trade-off points (sections III-D and III-E),

3) *the explanation of trade-off decisions* by describing which combinations of architectural tactics cause the trade-offs (section III-F).

First, software architects create reusable models specifying existing knowledge about architectural tactics and design alternatives in a specific domain or type of applications. These domain experts define feature models to represent tactics and design alternatives, and Feature-solution Graphs that maps the specified architectural tactics to the design alternatives that implement them.. The links between a tactic and its corresponding design alternatives may be *forces* or *prohibits* to indicate respectively, that the implementation of a tactic must include, or cannot include a particular design alternative.

Once this architectural design knowledge is codified into the mentioned models, the same or other architects can use them to select a set of architectural tactics for an application. FaMoSA, leveraging the information in the FS-Graph, specializes the feature model for design alternatives according to the tactics selected by the architect. FaMoSA detects when the selected tactics produce a set of conflicting design alternatives. These conflicts are reported as trade-off points that software architects must review and solve.

Finally, FaMoSA can automatically explain the trade-off decisions by describing the combination of tactics that caused the trade-off points.

B. Modeling Architectural tactics and design alternatives

In FaMoSA, architectural tactics and design alternatives are modeled as feature models. A *Feature Model* is a compact representation of the commonalities and variabilities in a domain or a set of products [8]. Originally aimed to analyse and document a family of products, these models allow engineers to define constraints about which combinations of features can be part of a product. A *Feature Model Configuration* is a combination of the features in a feature model. A configuration is considered valid when it includes a set of features that satisfies all the constraints defined in the model. The semantics of a feature model is commonly defined as its set of valid configurations [14].

Definition 1 (Feature Model): A feature model *fm* is a tuple $fm = (F, r, DE, w, \lambda, \Phi)$ where F is a (non empty) set of features, $r \in F$ is the root, λ assigns a decomposition operator to each feature, w assigns types (i.e., *Optional*, *Mandatory*, or *NonSelectable*) for each feature, and Φ is the set of constrains among the features (i.e., cross constraints).

Definition 2 (Architectural tactics and design alternatives feature models): In FaMoSA,

1) an Architectural Tactic Feature model fm_{AR} is a feature model that represents architectural tactics, and
2) a Design Alternative Feature model fm_{DE} is a feature model that represents design alternatives for a domain or type of applications.

Considering the preceding definitions, when software architects decide on architectural tactics or design alternatives, they are selecting features for one of the above mentioned feature models.

Definition 3 (Feature Configuration): A Configuration c for a feature model *fm* is a set of features f such that $f \subseteq F$.

- *(valid configuration)*. A configuration c is valid if c satisfies all the constraints defined in *fm*, i.e.,. $c \models fm$.
- *(feature model semantics)*. The semantics of *fm*, noted as $[\![fm]\!]$ is the set of all the valid configurations.

Definition 4 (Architectural tactics and design alternatives configurations): In FaMoSA, for an specific application, a software architect can define

1) an Architectural Tactic Configuration c_{AR} as a selection of architectural tactics, and
2) a Design Alternative Configuration c_{AR} as a selection of design alternatives.

C. Relating architectural tactics with design alternatives

We use Feature-Solution Graphs to specify correspondences between tactics and design alternatives. A *Feature-Solution Graph* is a set of directed relations between feature models allowing the automatic derivation of a target configuration from a source configuration [11]. In FaMoSA, these graphs allow the derivation of a configuration of design alternatives from a given a configuration of architectural tactics.

In this paper, we are considering FS-Graphs using three kinds of relations:

forces relations. We say that a tactic t forces a design alternative d when the application of the tactic t requires the use of the design d. For instance, in the above presented tactics (Section II), the application of the *maintain copies of data* tactic requires the use of the *entity cache* design alternative.

suggests relations. A tactic t suggests a design alternative d when the application of tactic t implies the use of the design alternative d only when a design alternative of the same type is required. For instance, in the example, the *reduce overhead* tactic suggests the use of a *sticky-session load balancer* when a *multi-node* web tier is used.

prohibits relations. A tactic t prohibits a design alternative d when the application of tactic t implies the no use of the design alternative d. For instance, the *reduce overhead* tactic in the example prohibits the use of *pessimistic locking* in the business logic layer.

Definition 5 (Feature-Solution Graph): A feature-solution graph *fsg* is a tuple $fsg = (fm, fm', SUG, FOR, PRO)$ where fm and fm' are the left-side and right-side feature models. We write F to refer to the set of features in *fm* and F' to the set of features in fm'. The $FOR \subseteq F \times F'$, $SUG \subseteq F \times F'$ and $PRO \subseteq F \times F'$ are the sets of *forces*, *suggests* and *prohibits* relations respectively.

Definition 6 (Architectural Feature-Solution Graph): In FaMoSA, an architectural feature-solution graphs $fsg_{AR_2DE} = (fm_{AR}, fm_{DE}, SUG, FOR, PRO)$ relates tactics in fm_{AR} to design alternatives in fm_{DE}

D. Determining design alternatives for a set of tactics

Once an architect decide about tactics, we use the FS-Graphs to specialize the design alternatives feature model. An *specialization process* is a transformation that takes a feature model as input and yields another feature model as output, such that the set of the valid configurations denoted by the output model is a subset of the valid configurations denoted by the former diagram [15]. In FaMoSA, the relations in the FS-Graph are used to determine a feature model, i.e., to a subset of design alternatives, that implements the tactics specified in an architectural tactic configuration.

Algorithm 1 describes how the feature model for design alternatives is specialized. Basically, given a FS-Graph fsg and a configuration c that is valid regarding the source FM in fsg, the relations in fsg represent the following modifications:

forces relations. A forces relation $f \xrightarrow{forces} f'$ in fsg denotes that the types of features in fsg must be m f' in the right-side FM of fsg must be converted to full-mandatory when f is included in the configuration c.

suggests relations. A suggests relation $f \xrightarrow{suggests} f'$ in fsg indicates that the feature f' in the right-side FM of fsg must be typed as mandatory just in the feature group where it belongs when f is included in c.

prohibits relations. A prohibits relation $f \xrightarrow{prohibits} f'$ denotes that the feature f' in the right-side FM of fsg must be typed as a non selectable feature when f is included in c.

Algorithm 1 Specialization of the right-side FM

```
1:  procedure APPLYFSG($c, fsg$)
2:      for all $f \in c$ do
3:          for all $(f, f') \in FOR$ do
4:              $makeFullMandatory(f')$
5:          end for
6:          for all $(f, f') \in SUG$ do
7:              $makeMandatory(f')$
8:          end for
9:          for all $(f, f') \in PRO$ do
10:             $makeNonSelectable(f')$
11:         end for
12:     end for
13:     return $fsg$
14: end procedure
```

Definition 7 (Determining design alternatives for tactics): Given an architectural feature-solution graph fsg_{AR_2DE} that relates tactics in fm_{AR} to design alternatives in fm_{DE}, the corresponding design alternatives fm'_{DE} for a set of tactics c_{AR}, is the resulting model after specialization, i.e., the model in $fsg'_{AR_2DE} = (fm'_{AR}, fm'_{DE}, SUG, FOR, PRO)$ such that $fsg'_{AR_2DE} = applyFsg(c_{AR}, fsg_{AR_2DE})$

- For convenience, we write $fm'_{DE} = specialize(c_{AR})$ to denote the resulting feature model of design alternatives.

Definition 8 (Determining design alternatives for a set of tactics): After specializing the design alternatives feature model, the selected set of alternatives c'_{AR} is the set of full mandatory features in the resulting feature model fm'_{DE}, i.e., the selected design alternatives c'_{AR} is a configuration of the resulting design alternative feature model fm_{DE} such that $\forall c \in [\![fm_{DE}]\!] | c'_{AR} \subseteq c$

E. Detecting trade-offs

FaMoSA focuses on detecting trade-off points based on conflicts on the corresponding design alternatives. These conflicts are identified as inconsistencies in the resulting right-side feature model after the specialization process. A conflict exists if the resulting feature model is invalid. An *invalid feature model* is a feature model that does not have any valid configuration (i.e., has an empty semantic set) [15]. For instance, a feature model including full-mandatory features typed as non-selectable are invalid feature models because it is not possible to create a valid configuration including a feature, and without including the same feature.

Definition 9 (Valid Feature Model): A valid feature model fm is a model which semantic is not empty, i.e., $[\![fm]\!] \neq \emptyset$.

- *(Invalid feature model).* An invalid feature model fm is a model which semantic is empty, i.e., $[\![fm]\!] = \emptyset$.

Definition 10 (Tactics that lead to trade-offs): In FaMoSA, a set of tactics c_{AR}, leads to trade-offs if, after applying the FS-Graph fsg_{AR2DE}, the resulting specialized design alternative feature model fm'_{DE} is invalid, i.e., $[\![fm'_{DE}]\!] = \emptyset$. We also write $[\![specialize(c'_{AR})]\!] = \emptyset$

To determine if a feature model is invalid, there are many approaches[16]. Basically, these approaches take a feature model and transform it into a set of constraints that can be analysed using tools such as Constraint Programming (CSP) or SAT solvers. In addition, libraries such as FaMa[1] and SPLOT[2] provide APIs to define and validate feature models.

F. Explaining trade-offs

Trade-offs can be explained by describing the involved decisions, i.e., the sets of architectural tactics that lead to conflicts, and the *trade-off points*, i.e., the set of design alternatives involved in that conflicts.

Considering that conflicts are detected in the right-side of the FS-Graph, we first perform a *feature model diagnosis task* in the specialized feature model for design alternatives to determine the trade-off points. This task determines a *minimal diagnosis*, i.e., a minimal set of constraints which must be removed to make that feature model valid. Then, we use that diagnosis to determine 1) the trade-off points, i.e., the features involved in the conflict, and 2) the involved decisions, i.e., the selected architectural tactics that caused the conflict

Definition 11 (Configuration diagnosis): Given an invalid configuration c of a feature model fm, the set of features $d \subset c$ is a diagnosis if the configuration c' resulting of removing the

[1]http://www.isa.us.es/fama/

[2]http://www.splot-research.org/

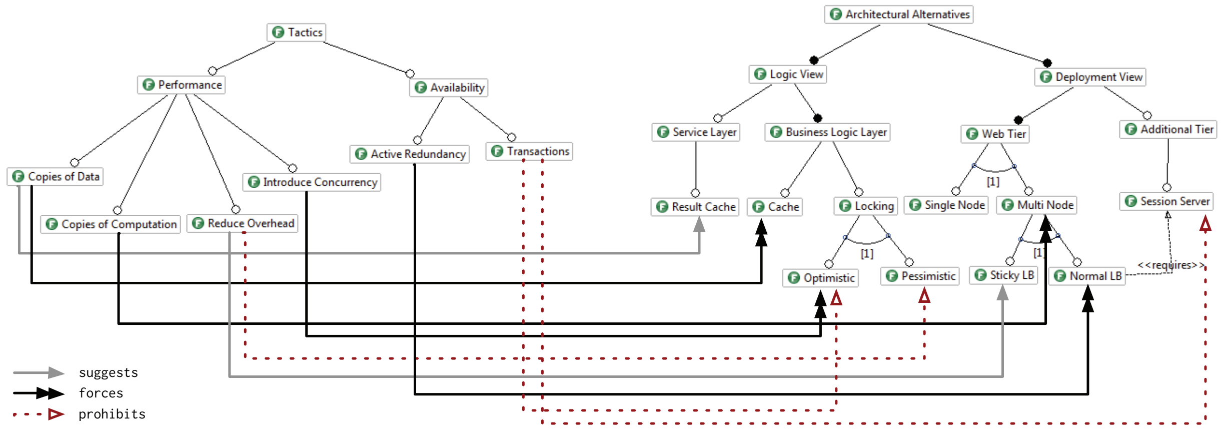


Fig. 2: FS-Graph of architectural tactics and design alternatives for database-based web applications.

diagnosis is valid, i.e., d is diagnosis if $\exists c' = c - d$ such that $c' \in [\![fm]\!]$

- *(minimal configuration diagnosis)*. A diagnosis d is minimal if there is not a subset $d' \subset d$ that is also a diagnosis.

Definition 12 (Trade-off points): Given a configuration of tactics c_{AR} that leads to a trade-off, the trade-off points are the subset d_{DE} of the corresponding set of design alternatives c_{DE} that is the minimal configuration diagnosis, i.e., d_{DE} contains the trade-off points if $d_{DE} \in \mathbb{D}_{DE} = \{d | \exists c' = c - d \bullet c' \in [\![fm_{DE}]\!]\} \wedge \nexists d' \in d_{DE} | d' \in \mathbb{D}_{DE}$

Definition 13 (Trade-off causes): Given a configuration of tactics c_{AR} that leads to a trade-off, the trade-off causes are the subset of tactics $d_{AR} \in c_{AR}$ that cause the selection of the trade-off points d_{DE} in the design alternatives, i.e., d_{AR} is the trade-off cause if $c'_{AR} = c_{AR} - d_{AR} \wedge [\![specialize(c'_{AR})]\!] \neq \emptyset$

There are several options to determine the diagnosis. One possible approach is based on the *Hitting Set Directed Acyclic Graph (HSDAG)* algorithm [17]. Using this algorithm, when a conflict is detected, an iterative process must search for the diagnosis by removing subsets of the configuration until the conflict is resolved. However, this approach is inefficient. More recent algorithms, such as *QickXplain* [18] and *FastDiag* [19], are focused on more efficient solutions. Furthermore, modern tools for CSP and SAT solving offer efficient solutions to determine minimal diagnoses.

IV. Trade-off Management for Tactics in Database-Based Web Applications

This section illustrates how FaMoSA can be used to detect trade-off points and explain trade-off decisions using the tactics and alternatives described in section II.

A. Modeling Architectural tactics and design alternatives

Figure 2 shows the models that capture the architectural tactics and design alternatives in the example: 1) *tacticsFm*, an Architectural tactics feature model previously presented in Figure 1a; 2) *designFm*, a Design Alternative feature model previously presented in Figure 1b; and 3) and *tactics2designFsg*, a FS-Graph that relates tactics in the former to features in the latter as defined in table I.

B. Determining design alternatives for a set of tactics

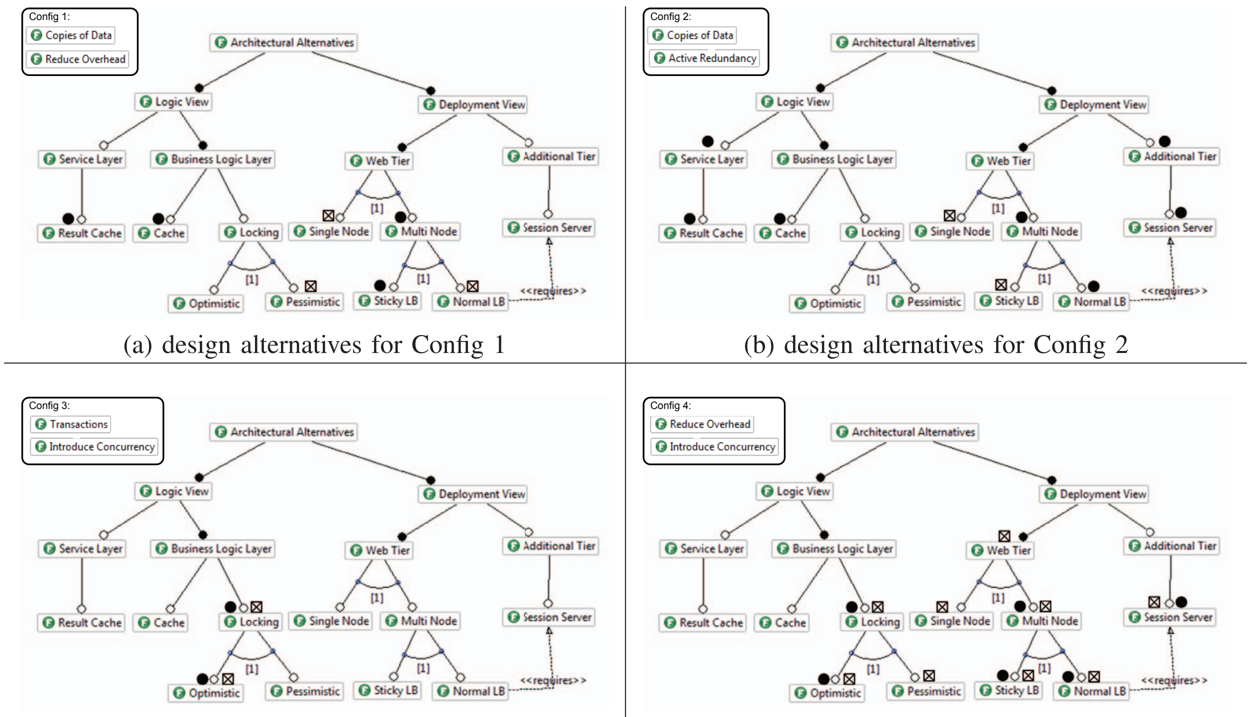
To illustrate the analysis, we consider four different combinations of tactics: 1) *config1*, that combines the *maintain copies of data* and *reduce overhead* performance tactics; 2) *config2*, that combines the *maintain copies of data* tactic for performance with the *active redundancy* for availability. 3) *config3*, combining the *introduce concurrency* tactic for performance with the *transaction* for availability, and 4) *config4*, with the *introduce concurrency* and *reduce overhead* tactics for performance with the *active redundancy* and *transaction* for availability.

FaMoSA have been used to determine the design alternatives that correspond to each configuration (i.e. set of tactics). We have two implementations[3]: an implementation in Alloy, used to verify some properties of our approach, and a Java-based implementation that rely on FaMa and SPLOT libraries to process feature models. After using our prototypes, FaMoSA detects conflicts in configs 3 and 4, while configs 1 and 2 are valid.

Figure 3 shows the results of specializing the feature model of design alternatives for each configuration. It shows the original feature model of design alternatives (figure 1a) and a set of marks over each box representing a feature showing the types that resulted after the specialization process : A ○ indicates that a feature has the Optional type, ● that has the Mandatory type, and ☒ the Non-selectable type. Note that a feature can have several types at the same time.

a) Design alternatives for config1: Selecting the tactics *maintain copies of data* and *reduce overhead* results in an specialized feature model where some features have been selected by the tactics and includes the mandatory type in addition to their type in the original model: the *maintain copies of data* includes the mandatory type to the *result cache*

[3]http://soft.vub.ac.be/~jchavarr/famosa



(a) design alternatives for Config 1

(b) design alternatives for Config 2

(c) design alternatives for Config 3

(d) design alternatives for Config 4

○ represents that a feature has an optional type, ☒ a non-selectable type and ● a mandatory type

Fig. 3: Result of specialization process for each example configuration

and *cache* alternatives, and the *reduce overhead* includes the mandatory type to the *sticky session load balancer*, and the *non-selectable* type to the *pessimistic locking* and *normal load balancer* alternatives. The resulting feature model does not include conflicts. Note that selecting *sticky session load balancer* adds the *non-selectable* type to the *sticky-session load balancer*. In addition, the inclusion of *mandatory* to *multi-node* web tier adds the *non-selectable* type to *single node* web-tier. This means that a valid configuration of the specialized design alternatives cannot include these design alternatives. In addition, note that software architect can make additional decisions with the resulting feature model. For instance, an architect can choose (or not) to use *optimistic locking*.

b) Design alternatives for config2: Selecting *maintain copies of data* and *active redundancy* also results in a set of design alternatives without conflicts. Note that the *maintain copies of data* tactic add the *mandatory* type to *result cache*. However, because the tactics suggests and do not forces that design alternative, the service layer is not *mandatory*. That means that the architect can select (or not) to include a service layer to the application. Note that selecting *normal load balancer* adds the *non-selectable* type to the *sticky-session load balancer*. In addition, the inclusion of *mandatory* to *multi-node* web tier adds the *non-selectable* type to *single node* web-tier.

C. Detecting and explaining Trade-off points

For the rest of the example configurations, there are conflicts in the resulting feature model of design alternatives, i.e., there are trade-off points that must be solved.

c) Trade-off for config3: The specialized feature model that correspond to *introduce concurrency* and *transaction* tactics has conflicts and is invalid. Basically, there is a conflict identifiable in the *optimistic locking* alternative. While the *introduce concurrency* tactic *forces* the use of *optimistic locking*, the *transaction prohibits* it. In the figure, it is represented by a full mandatory feature that has and the type non-selectable.

d) Trade-off for config4: Application of the *introduce concurrency*, *reduce overhead*, *active redundancy* and *transactions* tactics result in a set of conflicts. Some of them are directly identifiable by the relations in figure 3, but other are identifiable by the propagation of changes. For instance, here the *active redundancy* tactic add the *mandatory* type to the *normal load balancer* and the *transactions* tactic add the *non-mandatory* type to *session server*. This results in a conflict because there is a *requires* relation between the *normal load balancer* and the *session server*.

FaMoSA is able to detect and explain these trade-offs by processing the resulting feature models where the trade-offs are represented as conflicts.

V. Related Work

Although the idea of using feature models to select architectural tactics is not new, the idea of using them to detect trade-offs early has been not explored before. In this section, we summarizes related work.

Feature models to select tactics and design alternatives: Kim et al. [13] proposed *architectural tactic feature models* to describe possible combinations of architectural tactics. However, they consider tactics as UML-based model refactorings and do not consider tactic implementations that would lead to a trade-off. Our work extends their approach in order to specify how the tactics are implemented using design alternatives and to detect trade-offs.

Approaches such as COVAMOF [20] and QADA [21] use feature models to specify alternative requirements about quality attributes or technological platforms, and to relate them to existing alternatives of components and assets. These models are used by software architects or by automated processes to determine for a set of requirements, the corresponding design alternatives and components. None of these approaches support detection of trade-off points, nor the explanation of trade-off decisions.

Trade-off management: Approaches such as ATAM [3] and CBAM [4] include activities to determine trade-off points, and explain trade-off decisions. However, in contrast to our approach, they require a complete architectural design as an input. Furthermore, detection and explanation of trade-offs is performed by hand, in peer review sessions.

Other approaches use search-based techniques to detect and propose alternatives in trade-offs related to performance [7] and costs [22]. In contrast, FaMoSA focus on early detection and does not require complete or partial architectural designs. We consider that all these approaches, including ours, complement each to the others.

VI. Conclusions

We have presented FaMoSA, our approach to detect trade-off points and explain trade-off solutions based on feature models and feature solution graphs. It uses an specialization process to determine, for a set of tactics, the corresponding design alternatives, the conflicts between these alternatives, and the tactics which caused the conflicts.

FaMoSA detects trade-off points and explains trade-off decisions based on the constraints specified in the FS-Graphs. Thus, it cannot detect trade-offs caused by other constraints or by interactions only detectable at runtime. A future work is the integration of our approach to others that use attributes in feature models and simulation models to determine trade-offs.

Several configuration scenarios such as multi-level staged configuration, multi-view feature models and workflow-based configuration processes use processes for feature model specialization. Future work is planned on evaluating our approach to detect conflicts and propose fixes in these scenarios.

VII. Acknowledgments

Jaime Chavarriaga is a recipient of a COLCIENCIAS fellowship. Carlos Noguera is funded by the AIRCO project of the Fonds Wetenschappelijk Onderzoek.

References

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley Professional, 2012.

[2] R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord, and W. Wood, “Attribute-Driven Design (ADD), version 2.0. (CMU/SEI-2006-TR-023),” Software Engineering Institute, Tech. Rep., 2006.

[3] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, “The architecture tradeoff analysis method. (CMU/SEI-98-TR-008),” Software Engineering Institute, Tech. Rep.

[4] R. Nord, M. Barbacci, P. Clements, R. Kazman, M. Klein, and J. Tomayko, “Integrating the Architecture Tradeoff Analysis Method (ATAM) with the Cost Benefit Analysis Method (CBAM). (CMU/SEI-2003-TN-038),” Software Engineering Institute, Carnegie Mellon University, Tech. Rep., 2003.

[5] M. Svahnberg and K. Henningsson, “Consolidating different views of quality attribute relationships,” in *Software Quality, 2009. WOSQ ’09. ICSE Workshop on*, 2009, pp. 46–50.

[6] N. B. Harrison and P. Avgeriou, “Implementing reliability: The interaction of requirements, tactics and architecture patterns,” in *WADS*, 2010, pp. 97–122.

[7] A. Koziolek, H. Koziolek, and R. H. Reussner, “Peropteryx: Automated application of tactics in multi-objective software architecture optimization,” in *7th International Conference on the Quality of Software Architectures (QoSA 2011)*, 2011, pp. 33–42.

[8] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, “Feature-oriented domain analysis (FODA) feasibility study. (CMU/SEI-90-TR-021),” Software Engineering Institute, Tech. Rep., 1990.

[9] K. Czarnecki and U. W. Eisenecker, *Generative Programming*. Addison-Wesley, 2000.

[10] H. Bruin and H. Vliet, “Scenario-based generation and evaluation of software architectures,” in *Third International Conference in Generative and Component-Based Software Engineering (GCSE 2001)*, 2001, pp. 128–139.

[11] J. Chavarriaga, C. Noguera, R. Casallas, and V. Jonckers, “ supporting multi-level configuration with feature-solution graphs: formal semantics and alloy implementation,” Vrije Universiteit Brussel, Tech. Rep., 2013.

[12] E. Bagheri, T. D. Noia, D. Gasevic, and A. Ragone, “Formalizing interactive staged feature model configuration,” *Journal of Software Maintenance and Evolution*, vol. 24, no. 4, pp. 375–400, 2010.

[13] S. Kim, D.-K. Kim, L. Lu, and S. Park, “Quality-driven architecture development using architectural tactics,” *Journal of Systems and Software*, vol. 82, no. 8, pp. 1211 –1231, 2009.

[14] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux, “Feature diagrams: A survey and a formal semantics,” in *14th IEEE International Requirements Engineering Conference (RE 2006)*, 2006, pp. 139–148.

[15] K. Czarnecki, S. Helsen, and U. Eisenecker, “Staged configuration through specialization and multilevel configuration of feature models,” *Software Process: Improvement and Practice*, vol. 10, no. 2, pp. 143–169, 2005, ISSN: 1099-1670.

[16] D. Benavides, S. Segura, and A. Ruiz-Corts, “Automated analysis of feature models 20 years later: A literature review,” *Information Systems*, vol. 35, no. 6, pp. 615 –636, 2010.

[17] R. Reiter, “A theory of diagnosis from first principles,” *Artificial Intelligence*, vol. 32, no. 1, pp. 57–95, Apr. 1987.

[18] U. Junker, “Quickxplain: Preferred explanations and relaxations for over-constrained problems,” in *19th National Conference on Artifical Intelligence (AAAI’04)*, 2004, pp. 167–172.

[19] A. Felfernig, M. Schubert, and C. Zehentner, “An efficient diagnosis algorithm for inconsistent constraint sets,” *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 26, pp. 53–62, 01 Feb. 2012.

[20] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch, “Covamof: A framework for modeling variability in software product families,” in *In Proceedings of the Third International Software Product Line Conference (SPLC 2004)*, 2004.

[21] M. Matinlassi, “Quality-driven software architecture model transformation,” in *Fifth Working IEEE / IFIP Conference on Software Architecture (WICSA 2005)*, IEEE Computer Society, 2005, pp. 199–200.

[22] J. García-Galán, O. F. Rana, P. Trinidad, and A. R. Cortés, “Migrating to the cloud - A software product line based analysis,” in *3rd International Conference on Cloud Computing and Services Science (CLOSER 2013)*, 2013, pp. 416–426.