
Amanda S. Nascimento  
Institute of Computing  
University of Campinas  
Campinas, SP, Brazil  
asilva@ic.unicamp.br

Cecília M.F. Rubira  
Institute of Computing  
University of Campinas  
Campinas, SP, Brazil  
cmrubira@ic.unicamp.br

Fernando Castor  
Informatics Center  
Federal University of Pernambuco  
Recife, PE, Brazil  
castor@cin.ufpe.br

Abstract—Fault tolerance solutions based on design diversity have been adopted to mediate the communication between clients and functionally equivalent services, i.e., alternate services, to tolerate software faults in applications based on Service-Oriented Architectures (SOAs). However, there is a growing need for solutions for fault-tolerant SOAs that are able to adapt themselves in response to dynamic changes in client requirements, resource constraints and quality of reused alternate services (QoS). Over recent years, software engineers have been evaluating the benefits of both Software Product Line (SPL) and self-adaptive systems by adopting feature models to reason about runtime variability. In our previous work, we presented a preliminary QoS-aware and self-adaptive framework to support a family of software fault tolerance strategies applied to SOA. The framework employed a feature-based approach for runtime adaptation and self-configuration through dynamic binding of variants (i.e., variable features). Nevertheless, the conceptual model of adaptation upon which the framework was built was represented implicitly in the form of domain knowledge. In this work, we report our experience on using a well-established set of modelling dimensions to classify and explicitly describe aspects of the proposed framework that are relevant for self-adaptation, to known: (i) goals or objectives associated with the adaptation; (ii) causes for adaptation; (iii) mechanisms related to the adaptation process itself; and (iv) effects of adaptation upon the framework. The classification suggests directions for future work on how to improve the self-adaptation process of the preliminary framework.

Keywords-Dynamic Software Product Line; Dependability improvement; Autonomic Control Loop; QoS-Aware; Self-adaptive systems;

I. INTRODUCTION

The adoption of software fault tolerance techniques based on design diversity has been advocated as a means of coping with residual software design faults in operational software [1]. Design diversity is the provision of functionally equivalent software components, called alternates, through different design and implementations [1]. In summary, three major design issues need to be considered while building fault-tolerant architectures based on design diversity, namely, (i) selection of alternates that are sufficiently diverse and able to tolerate software faults; (ii) execution of alternates; and (iii) selection of an adjudicator to determine the acceptability of the results obtained from the alternate [1], [2], [3]. Each design issue can be realized by a set of alternative design decisions, which, in turn, imply in different degrees of quality requirements (e.g., memory consumption, financial cost, response time and reliability). For example, alternates might be executed either sequentially or in parallel and alternate outputs might be adjudicated by adopting different voting and acceptance algorithms [1], [2].

In the context of Service-Oriented Architectures (SOA), on the web, a number of functionally equivalent web services, i.e. alternate services, exists to achieve a particular task [4]. Nevertheless, for applications comprising services, usually controlled by third parties [5], it is necessary to tolerate software faults stemming from design errors or implementation mistakes [1], [6]. Due to the low cost of reusing existing alternate services, several diversity-based approaches have been developed to support reliable SOA-based applications. These approaches operate as mediator between clients and alternate services. The latter are structured in fault-tolerant composite web services [6], [4], called FT-compositions for simplicity. From the clients’ viewpoint, the FT-composition works as a single, reliable service.

Nevertheless, existing FT-compositions have some drawbacks. First, some solutions do not deal with some of the above mentioned design issues (e.g., different adjudicators) [7], [8]. Secondly, applications based on SOA rely on a singular scenario, where the environment is highly dynamic and several decisions should be postponed until runtime, the control is highly distributed and we usually have conflicting client requirements. Moreover, some guarantees specified by QoS attributes of the reused alternate services may become invalid at runtime [9], [5]. Therefore, FT-compositions should be able to dynamically reconfigure themselves to cope with dynamic changes of (i) client requirements [9], [5], and (ii) the level of quality of services (QoS). Current FT-compositions are mainly static approaches and support limited self-adaptation capabilities [10], [7].

Software product lines (SPLs) and adaptive systems support
software variability to cope with changing requirements [11]. Variability is the capacity that a software system or a software artefact has to be modified, customized and configured for use in a particular context at some point in its life-cycle, including, at runtime [12]. Variability can be described in terms of features, which are central for development and configuration of SPLs [13]. A special kind of SPL asset is the Product Line Architecture (PLA), which represents the SPL reference architecture [14], [12]. Self-adaptive systems can be built based on software product line principles by means of feature-based runtime adaptation [15], [16]. In this way, feature models are used to describe dependencies between features and also to reason about runtime variability [17], [18].

To bridge the above mentioned gaps in existing FT-compositions, in our previous work [6], we presented our preliminary solution of a QoS-aware and self-adaptive framework to support a family of closely related fault tolerance strategies applied to applications based on SOA. Our solution relies on key activities of the autonomic control loop (i.e. collect, analyse, plan and execute) to support dynamic management of software variability. The framework decides and instantiates at runtime a particular fault tolerance strategy to be executed in close accordance with clients requirements, current context and high-level policies. In this work, we explicitly describe our framework’s self-adaptive behaviour. For that, we adopted the modelling dimensions proposed by Andersson et al. [19]. The modelling dimensions, which are part of a conceptual model, describe various facets of the system that are relevant to self-adaptation and are classified in terms of four groups, to know: (i) goals - goals are objectives the system under consideration should achieve; (ii) change - change is the cause for adaptation; (iii) mechanisms - what is the reaction of the system towards change; (iv) effects - what is the impact of adaptation upon the system. The conceptual model [19] was used to (i) achieve a better understanding about the specifics and intrinsically hidden self-adaptive properties of our preliminary solution for FT-compositions; and (ii) select suitable solutions to support the self-adaptation.

It is important to emphasize that Andersson et al. [19] evaluated the modelling dimensions by applying them to several previously developed self-adaptive software systems belonging to different classes of application domains: (1) Traffic Jam Monitoring Systems, (2) Embedded Mobile Systems and (3) High Performance Computing and Sensor Networks [19]. In this work, we report our experience in using the taxonomy by Andersson et al.[19] to a different application domain, for instance for self-adaptive and QoS-aware mechanisms for FT-compositions. Therefore, our experience report contributes to strengthening a common set of vocabulary for specifying the self-adaptive properties under consideration, that is, this work contributes to establishing a baseline from which key aspects of different self-adaptive system can be easily identified and compared [19].

II. BACKGROUND

In this section, we present an overview of software fault-tolerant architectures; synergies between software product lines and self-adaptive systems; and an overview of the modelling dimensions for self-adaptive systems proposed by Andersson et al [19].

A. Software Fault-Tolerant Architectures

A fault is the identified or hypothesized cause of an error [20], [21]. An error is part of the system state that is liable to lead to a failure [20], [21]. A failure, in turn, occurs when the service delivered by the system deviates from the specified service [1], [21]. Software failures can be avoided by utilizing fault-tolerant architectures in order to detect errors when they occur and react appropriately [1]. Error recovery is performed using either backward recovery or forward recovery. On one hand, backward recovery attempts to return the system to a correct or error-free state by restoring or rolling back the system to a previously saved state, which is assumed to be error-free [1]. On the other hand, forward recovery attempts to return the system to a correct or error-free state by finding a new state from which the system can continue operation. Compared with backward error recovery, the forward recovery is usually more efficient in terms of the overhead (e.g. time and memory) it requires [1]. In particular, this review focuses on software fault-tolerant architectures based on design diversity that employ forward error recovery.

Figure 1 illustrates the basic design diversity concept. Inputs are distributed to multiple software components, each with equivalent functionality yet different designs, called variants. The variants execute their operations and produce their results. Potentially, several alternate results can be produced, from which a single correct or acceptable result must be derived [1]. The mechanism responsible for this task is called an adjudicator, or decision mechanism.

Three major design issues need to be considered while building fault tolerance solutions based on design diversity that leverage forward error recovery, namely, selection of variants; variant execution schemes; and judgement on results acceptability [1], [3], [2]. Each design decision, in turn, may be realised by a set of alternative design solutions. For example, (i) variants can be chosen at different points during

![Fig. 1. Basic Design Diversity](image-url)
the software life-cycle - they may be chosen at design time by the engineer, configured manually once the software is deployed, or even be discovered and selected at run-time by the software itself; (ii) variants can be executed either sequentially or in parallel; and (iii) by the software itself;

(iii) deployed, or even be discovered and selected at run-time by the engineer, configured manually once the software is run.

The softw are life-cycle - they may be chosen at design time by the different possibilities that exist to satisfy a variation point. In a complementary way, variants are decision can be made. In a complementary way, variants are point is the place at the software architecture where a design which are described through variation points. A variation feature model represents the commonalities among all products from a set of core assets for a given domain, exploiting the commonalities and variabilities among these products [14], [12]. Based on a hierarchical structure, a feature model represents the commonalities among all products of a product line as mandatory features, while variabilities among products are represented as variable features, which are also called variants. Variable features extensively fall into three categories: (i) optional, which may or may not be present in a product; (ii) alternative, which indicates a set of features, from which only one must be present in a product; and (iii) multiple features, which represents a set of features, from which at least one must be present in a product.

A key factor for successfully implementing an architectural product line approach is to structure variabilities into a product line architecture (PLA) in terms of variable architectural elements, and their respective interfaces, which are associated with variants [12]. In PLAs, software variability can be reached by delaying certain architectural design decisions, which are described through variation points. A variation point is the place at the software architecture where a design decision can be made. In a complementary way, variants are the different possibilities that exist to satisfy a variation point [14]. Binding the variant is the selection of some variant supported by a variation point. Therefore, the product line infrastructure consists of the common architecture and its variation points; a collection of reusable parts that fit into the architecture; and the decision model, which identifies variability and commonality in domain requirements and how requirements choices map to choices of parts [15].

Self-adaptive systems can be built based on software product line principles by means of feature-based runtime adaptation. From the perspective of runtime adaptation, well-established variability modelling in the SPL domain promises to be a valuable basis for the definition of appropriate models at runtime [16]. Therefore, although variability analysis and design can be performed at development time, the variability binding and reconfiguration is performed at runtime. Thus it requires some kinds of variability mechanisms to map high-level variations, which are represented by features, to low-level implementation, in the PLA level, and support runtime reconfiguration [22]. As a result, the self-adaptation strategies can be obtained and specified in a higher feature level rather than the lower program level, which makes it easily validated and understood by the system users [22].

B. Synergies between Software-Product Line and Self-Adaptation Systems

Software product lines (SPLs) and self-adaptive systems aim at variability to cope with changing requirements [15], [16]. In traditional SPLs, features are bound statically before runtime. By contrast, self-adaptive systems support feature binding at runtime [11]. Software product line is a systematic software reuse approach that promotes the generation of specific products from a set of core assets for a given domain, exploiting the commonalities and variabilities among these products [14], [12]. Feature modelling is one of the most accepted ways to represent commonalities and variabilities at the requirements phase. A feature is a system property that is relevant to some stakeholder [14], [12]. Based on a hierarchical structure, a feature model represents the commonalities among all products of a product line as mandatory features, while variabilities among products are represented as variable features, which are also called variants. Variable features extensively fall into three categories: (i) optional, which may or may not be present in a product; (ii) alternative, which indicates a set of features, from which only one must be present in a product; and (iii) multiple features, which represents a set of features, from which at least one must be present in a product.

A key factor for successfully implementing an architectural product line approach is to structure variabilities into a product line architecture (PLA) in terms of variable architectural elements, and their respective interfaces, which are associated with variants [12]. In PLAs, software variability can be reached by delaying certain architectural design decisions, which are described through variation points. A variation point is the place at the software architecture where a design decision can be made. In a complementary way, variants are the different possibilities that exist to satisfy a variation point [14]. Binding the variant is the selection of some variant supported by a variation point. Therefore, the product line infrastructure consists of the common architecture and its variation points; a collection of reusable parts that fit into

C. Modelling Dimensions of Self-Adaptive Systems

Modelling dimensions proposed by Andersson et al. [19] for self-adaptive systems allows engineers to precisely specify the self-adaptive properties under consideration. In the following, we present the dimensions in term of four groups, (i) self-adaptability aspects of the system goals; (ii) causes of self-adaptation; (iii) the mechanisms to achieve self-adaptability; and (iv) the effects of self-adaptability upon a system [19]. We summarize some details of the modelling dimensions in Table I [19].

III. A CONCEPTUAL MODEL OF THE QoS-AWARE AND SELF-ADAPTIVE FRAMEWORK FOR FT-COMPOSITIONS

In this section, we present an overview of our preliminary solution for FT-compositions and describe its relevant properties related to the adaptation process.

A. An Overview of the Self-Adaptive Framework

Our preliminary QoS-aware and self-adaptive framework for FT-compositions supports a family of fault tolerance strategies based on design diversity and forward error recovery [6]. The framework operates as mediator between service clients and alternate services and is able to dynamic instantiate a fault tolerance strategy according to the current context and high level policies. The proposed framework encompasses a feature model, a product line architecture, a decision model, artefacts implementing common and variable features and an autonomous controller. For instance, the feature model and product line architecture captures the commonalities and variabilities among largely adopted design diverse software fault tolerance techniques (e.g. N-Version Programming, N-Check Programming, Consensus Recovery Block and Acceptance Voting) and adjudicators (e.g. different voting and acceptance tests algorithms). All fault tolerance strategies identified (i.e. products from the software product line) must support all design issues related to a fault-tolerant architecture based on software design diversity (Section II-A). Therefore,


<table>
<thead>
<tr>
<th>Dimensions</th>
<th>Degree</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Evolution</td>
<td>Static to Dynamic</td>
<td>As the system as a whole evolves, the number of goals and the goals themselves may also change. In static goal evolution, changes are not expected. In the dynamic one, goals can change at run-time.</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Rigid, Constrained, Unconstrained</td>
<td>This dimension is related to the level of uncertainty associated with the goal specification. A goal is rigid when it is prescriptive (e.g. 'the system shall do this-'). A goal is unconstrained when its statement provides flexibility for dealing with uncertainty (e.g. 'the system might do this-'). Related to the constrained goal, there is flexibility as long as certain constraints are satisfied (e.g. 'the system may do this - as long as it does this -').</td>
</tr>
<tr>
<td>Duration</td>
<td>Temporary to Persistent</td>
<td>This dimension is concerned with the validity of a goal throughout the system's lifetime. A persistent goal should be valid throughout the system's lifetime. On the other hand, a temporary goal (e.g. short, medium and long term) may be valid for a period of time.</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>Single to Multiple</td>
<td>This dimension is related to the number of goals associated with the self-adaptability aspects of a system. In this way, a system can either have a single goal or multiple goals.</td>
</tr>
<tr>
<td>Dependency</td>
<td>Independent to Dependent (complementary to conflicting)</td>
<td>This dimension captures how the goals are related to each other in case a system has multiple goals. A system can have several goals such as they don't affect each other, i.e., they are independent. By contrast, several dependent goals can either be conflicting or they can be complementary with respect to the objectives that should be achieved. Tradeoffs have to be analyzed for identifying an optimal configuration of dependent conflicting goals.</td>
</tr>
</tbody>
</table>

| Change      |        |            |
| Source      | External (environmental), Internal (application, infrastructure) | This dimension identifies the origin of the change, which can be either internal to the system, depending on the scope of the system or external to the system. |
| Type        | Functional, Non-functional, Technological | This dimension refers to the nature of change, i.e., functional, non-functional or technological (e.g. software and hardware aspects). |
| Frequency   | Rare to Frequent | This dimension is concerned with how often a particular change occurs. |
| Anticipation| Foreseen, Foreseeable, Unforeseen | This dimension captures whether change can be predicted ahead of time. There are different degrees of anticipation: foreseen (taken care of), foreseeable (planned for), and unforeseen (not planned for). |

| Mechanisms  |        |            |
| Type        | Parametric to Structural | This dimension captures whether adaptation is related to the parameters of the systems components or to the structure of the system. Adaptation can be parametric or structural, or a combination of these. |
| Autonomy    | Autonomous to Assisted (system or human) | This dimension identifies the degree of outside intervention during adaptation. At run-time there is no influence external to the system guiding how the system should adapt in the autonomous case. To the contrary, a system can have a degree of self-adaptability when it is assisted, either by human participation or another system. |
| Organization| Centralized to Decentralized | This dimension captures whether adaptation is done by a single component or distributed amongst several components (i.e. no single component has complete control over the system). |
| Scope       | Local to Global | This dimension deals with the impact of adaptation in terms of space and identifies whether the adaptation is localized or involves the entire system (e.g. different software components). |
| Duration    | Short, Medium, Long Term | This dimension deals with the impact of adaptation in terms of time and refers to how long the adaptation lasts. Time characteristics should be considered relative to the application domain. |
| Timeliness  | Best Effort to Guaranteed | This dimension captures whether the time period for performing self-adaptation can be guaranteed. |
| Triggering  | Event-Trigger to Time-Trigger | This dimension identifies whether the change that triggers adaptation is associated with an event or a time slot. This dimension is based on the general assumption that it is possible to control how and when the adaptation should react to a certain change. |

| Effects     |        |            |
| Criticality | Harmless, Mission-Critical, Safety-Critical | This dimension captures impact upon the system in case the self-adaptation fails. |
| Predictability | Non-Deterministic to Deterministic | This dimension identifies whether the consequences of self-adaptation can be predictable both in value and time. Therefore, predictability is associated with guarantees surrounding the system. |
| Overhead    | Insignificant to Failure | This dimension captures the negative of system adaptation upon the quality of services of the system. |
selection and execution of alternate services and means to adjudicate results obtained from the alternate services are mandatory features. The variable features, or variants, are composed, in summary, by the different design choices associated to these design issues (e.g. alternate services may be executed either in parallel or sequentially; different voting adjudicators) - Section II-A.

Each fault tolerance strategy, a member of the SPL, is obtained through design choices, for example, (i) alternative services providing a particular task are discovered at run-time, executed in parallel and their results are sent to a majority voter; (ii) alternative services providing a particular task are specified at design time, executed in parallel and their results are sent to a mean voter; (iii) alternative services providing a particular task are discovered at run-time, executed sequentially and their results are sent to an acceptance test mechanism; and so on. Different design choices imply in different measures of quality requirements (e.g. memory utilization, execution time, reliability, financial costs and availability) perceived by the client [1] (Section II-A). The differences among them make each possible fault tolerance strategy suitable for a particular situation. For example, given a critical real-time system, which is usually time-constrained, the framework should (i) adopt a parallel execution scheme for better response time if the sum of the response time values of the alternate services is greater than a threshold value, otherwise, alternate variants should be executed sequentially. Finally, the autonomous controller supports dynamic management of software variability.

B. Modelling Dimensions of the Target Framework

The modelling dimensions proposed by Andersson [19] (Table I) were applied to our preliminary solution for FT-compositions as described in the following.

Goals The framework has one single goal related to self-adaptability aspects: to support adaptation of the current fault tolerance strategy to accommodate changes in the context in accordance with high level policies. For instance, an adaptation corresponds to a product derivation at runtime.

- **Evolution**: Static - The already described goal will not change over the life time of the framework.
- **Flexibility**: Rigid - High-level policies specified to achieve the goals should have the Event-Condition-Action (ECA) form [23], [13]. An ECA rule includes triggering events, guarding conditions, current system configurations, and a set of reconfiguration steps. The idea is that when triggering events occur and a guarding condition holds, the system should perform the reconfiguration steps through dynamic binding of variants [15]. We emphasize that, although rules can also be added during execution, the rules themselves are specified and validated before being added to the framework in execution.
- **Duration**: Persistent - The goal of the framework is valid throughout the systems lifetime.
- **Multiplicity**: Simple - The framework supports a single goal, as already mentioned.
- **Dependency**: Dependent - There are no dependencies since there is a single goal.

Change For the self-adaptation process it is necessary to take into account that the framework relies on a dynamic component infrastructure, consequently, variable features can be installed, started, stopped, updated, and uninstalled without bringing down the whole framework (e.g. unforeseen adjudicators and schemes to execute variants might be included at run-time whenever they conform to the reusable interfaces identifying variation points). Under these circumstances, a software engineer is in charge of keeping the corresponding product line infrastructure up-to-date (Section II-B). Moreover, alternate services can appear and sometimes disappear at runtime. Therefore, the context, defined as any information which is computationally accessible and upon which behavioural variations depend [24], encompasses fluctuation in the level of quality of the alternate services (e.g. response time, reliability); changes in client requirements; availability of alternate services; the product’s current state and available variable features.

- **Source**: External (environment) and Internal (application) - Source of changes could be either external environment, such as situation when quality of alternate services is impacted severely due to instabilities in the communication links. Alternatively, the source of changes could be internal application, for example, variable features might be added at runtime or existing ones might be removed.
- **Type**: Non-functional and Functional - Since changes are related to the QoS of the alternate services, the type of change is non-functional. There are also functional changes which are related to the addition, updating and removing of alternate services and variable features.
- **Frequency**: Frequently - Changes occur constantly and are constantly monitored.
- **Anticipation**: Foreseen and Foreseeable - In systems based on SOAs, changes in quality of services and client requirements are the norm, rather than the exception (foreseen). Moreover, it is already known that variants realising variations points can be installed, started, stopped, updated, and uninstalled without bringing down the whole system (foreseeable).

Mechanisms The mechanism for self-adaptation should be based on dynamic management of software variability by means of a feature-based runtime adaptation. The reliable mediation requisition (sent by the client) triggers an event that is captured by the event handler, which activates the decision engine [6]. The decision engine matches facts and data. This matching process infers conclusions which may result in a derivation of a new configuration of the component PLA (Analyse and Plan). The new configuration is achieved by the Adaptation Engine through dynamic binding of software components implementing the target variants. For the adaptation process, it is specified (i) a mapping of features to the components that are used for implementation [25]; and (ii) a model that contains the relationships between the context's
• **Type:** Structural - The type of the self-adaptation is structural since configurations are changed and components and connectors are replaced in the product line architecture in order to bind variants and, consequently, instantiate a particular software fault tolerance strategy at runtime.

• **Autonomy:** Autonomous - There is no influence external to the system during the adaptation process itself.

• **Organization:** The self-adaptation of the framework comprises an autonomous controller that manages a loop of collection, analysis, planning and execution and is responsible for supporting dynamic binding of variants, hence the organization is centralized.

• **Scope:** Local and Global - The current architectural configuration of the framework determines the scope of the self-adaptation. For instance, it may be global if it involves the reconfiguration of the whole product line architecture to come up with a new fault tolerance strategy.

• **Duration:** Short term: The adaptation process should be completed in seconds.

• **Timeliness:** Best effort The time period required for performing the adaptation depends on several factors, such as the number of reliable requisitions, the number of rules, the number of available variants implementing variation points, the number of QoS concerns to be analysed. Given the short duration of the adaptation, best effort meets the required timeliness.

• **Triggering:** In order to avoid the framework from constantly reconfiguring itself, changes are analysed whenever the framework receives a requisition for a reliable mediation. After that, the framework decides accordingly which fault tolerance strategy to instantiate.

**Effects:** An SOA-based application will operate according to the specification in spite of faults from reused services.

• **Criticality:** Mission-critical, safety-critical: The level of criticality of adaptation process may be mission-critical or safety-critical, depending on the nature of the application based on SOAs that sends a requisition for a reliable mediation to the proposed framework.

• **Predictability:** Deterministic - Since the adaptation is based on rule-based approaches in the form of ECAs, the consequences of self-adaptation is predictable both in value and time.

• **Overhead:** Insignificant to Failure - The overhead associated with the framework adaptation should be insignificant. Nevertheless, the number of number of rules, alternate services implementing a particular task and software components implementing variants related to variation points can become unfeasibly large and the framework might cease to be able to deliver its services due to the high-overhead of running the self-adaptation process.

In this work, we have employed the modelling dimensions proposed by Andersson et al. [19] to classify our preliminary solution of a self-adaptive and QoS-aware framework for FT-compositions. The framework supports a family of software fault tolerance strategies and it is responsible for deciding instantiating at runtime a particular strategy to be executed in close accordance with clients requirements, current context and high-level policies.

Firstly, by means of the presented classification, we were able to explicitly state *when*, *what*, *how* and *where* to adapt our solution for FT-compositions. We also achieved a better understanding of the self-adaptive properties under consideration and selected suitable solutions to support the adaptation process. Since we adopt a feature model to reason about runtime variability, the self-adaptation strategy employed by the proposed framework was specified in a higher feature level rather than the lower program level. In this way, a feature model exploiting the commonalities and variabilities among largely adopted design diverse software fault tolerance techniques is used to describe dependencies between features and also to reason about runtime variability [17], [18], [25]. Therefore, elements of the product line infrastructure (Section II-B) are represented at runtime and compose the employed reconfiguration models. Moreover, for the adaptation process it is necessary to specify mappings between features to the components that are used for implementation and between contexts’ variation points and features. An adaptation to accommodate changes (e.g. in quality of reused services) corresponds to a product, *i.e.* a specific fault tolerance strategy, derivation at runtime.

The preliminary framework has one single goal (*i.e.* to instantiate a fault tolerance strategy among others in accordance with the current context and high level policies). Nevertheless, to achieve this general goal it might be necessary to reason about multiple QoS dimensions and, eventually, to resolve trade-offs among them [19] (e.g. the instantiation of a specific fault tolerance strategy might improve the overall reliability and result in an increase in the overall financial cost). In this way, the revised framework should support well-defined extension points in which utility functions can be included to map the trade-offs among several conflicting goals to a scalar value, which is then used by means of rules for making decisions about adaptation [19].

The goal associated with the adaptation is rigid in the way it is currently expressed since we have adopted event-condition-action (ECA) rules. According to Bencomo et al. [15], the obvious benefits of this strategy include adaptation behaviour analysis and validation at design time, and efficiency at runtime. In particular, it is necessary to guarantee at design time that rules are consistent with the variants in the feature model, its constraints and dependencies. On the downside, however, Bencomo et al. [15] claim that the adoption of ECA rules requires the identification of all possible configurations and adaptations at design time. Moreover, the number of rules can
become unfeasibly large. In this sense, to refine our preliminary solution for FT-compositions, we intend to investigate the benefits and drawbacks of using goal-based approaches instead of rule-based ones. Goal-based approaches adopt more abstract representations of the decision model and typically combine decision making based on the maximization of an objective function with explicit constraints to rule out invalid configurations [15]. Using goals avoids problems related to the enumeration of variants at design time, but at the cost of more runtime overhead [15].

Due to the dynamic and unpredictable nature of services [5], [9], the revised framework should support a continuous fault tolerance mechanism, which should be evolved dynamically. As mentioned, variable features (e.g. different schemes to execute variant services and adjudicate their results) should be installed, started, stopped, updated, and uninstalled without bringing down the whole framework. To support it, the revised framework has been developed according to the OSGi technology, which is a set of specifications that defines a dynamic component system for Java [26]. These specifications enable a development model where applications are (dynamically) composed of many different (reusable) components. The OSGi framework is dynamic. It can update bundles, or OSGi components, on the fly and services can come and go. This dynamic service model allows bundles to find out what capabilities are available on the system and adapt the functionality they can provide [26]. This makes code of the revised framework more flexible and resilient to changes. For instance, the framework implementation relies on Equinox - an Eclipse project that provides a certified implementation of the OSGi R4.x core framework specification [26]. The rule engine, in turn, has been developed by means of the JRuleEngine [27]. In our preliminary prototype we used Drools, however, we have faced problems in reusing the Drools software components into the OSGi infrastructure.

By means of the proposed classification, we were able to realize that the negative impact of framework adaptation upon the frameworks performance can, in fact, range from insignificant to system failure - the latter may happen due to some scaling problems (e.g. the number of rules can become unfeasibly large, thus implying in a high-overhead of running the self-adaptation). Second, if the adaptation process fails, the consequences for the service client can range from mildly annoying to great financial losses or the loss of human life. Therefore, for the revised framework, we should consider measures to mitigate this risk (e.g. the framework must be subjected to rigorous and extensive testing and debugging). Moreover, monitoring the system, especially when there are several different QoS properties of interest, has an overhead [19]. According to Andersson et al. [19], the amount of degradation in QoS due to monitoring could outweigh the benefits of improvements in QoS to adaptation. In this way, we should employ light-weight monitoring techniques in the revised framework.

V. RELATED WORK

Existing self-adaptive solutions for fault-tolerant SOAs do not explicitly described relevant facets of the target adaptation process [28], [29], [30], [31], [32]. The publications regarding existing solutions are written from different viewpoints and rely on different technical background for the specification of self-adaptive behaviours. As a result, sometimes it is hard to understand concerns on adaptation that are employed by these solutions. Our experience report represents a step stone in providing a unified view of conceptual and physical concepts concerned with self-adaptive behaviour of fault tolerance mechanisms applied to SOAs.

Buckley et al. [33] define a taxonomy of evolution based on the object of change (where), system properties (what), temporal properties (when), and change support (how), nevertheless unlike the approach proposed by Andersson et al. [19], the taxonomy by Buckele is not focused on run-time adaptation (change) of software. Salehie and Tahvildari[34] present a landscape of research in self-adaptive software by highlighting relevant disciplines and some prominent research projects. They also define a taxonomy for self-adaptive systems based on concerns of adaptation related to how, what, when and where to adapt, which are also supported by the taxonomy proposed by Andersson et al. [19]. Moreover, the taxonomy by Andersson et al. [19], unlike the one by Salehie and Tahvildari[34], have been evaluated through applying it to several previously developed self-adaptive software systems. Bradbury et al. [35] classify support from dynamic software architecture languages, nevertheless they do not focus on architecture specification and does not consider the goals, unlike the taxonomy we have adopted.

VI. CONCLUDING REMARKS

In our previous work, based on software product line principles, we proposed a framework to support adaptive fault tolerance in service oriented systems in order to cope with changes in the level of Quality of Services (QoS) and user requirements. The framework employed a feature-based approach for runtime adaptation and self-configuration through dynamic binding of variants (i.e. variable features). Nevertheless, the adaptation process was unclear and implicitly represented in the form of domain knowledge. In this work, in turn, we described our experience on using a well-established conceptual model to classify the self-adaptive properties of our preliminary solution. By means of this classification, we identified some contributions and drawbacks of the framework’s self-adaptive behaviour. We also discussed some initial decision making about designs, tools and middleware platforms that will be employed in the revised framework in future work. To the best of our knowledge, this is the first work describing a self-adaptive system that leverages feature-based runtime approach for runtime adaptation by means of the modelling dimensions. Therefore, the presented classification might be used further to systematically, or even qualitatively, compare similar approaches.
Acknowledgment

This research was sponsored by UOL (www.uol.com.br), through its UOL Bolsa Pesquisa program, project number 20120217172801. Cecília is supported by CNPq (305331/2009-4) and FAPEP (2010/00628-1). Fernando is supported by CNPq/Brazil (306619/2011-3), FACEPE/Brazil (APQ-0395-1.03/10 and APQ-1359-1.03/12), and by INES (CNPq 573964/2008-4 and FACEPE APQ-1037-1.03/08).

References


