Problem-Solution Feature Interactions as Configuration Knowledge in Distributed Runtime Adaptations

Frans SANEN, Eddy TRUYEN and Wouter JOOSEN
DistriNet, Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{frans.sanen,eddy.truyen,wouter.joosen}@cs.kuleuven.be

Abstract. Current generative programming approaches use configuration knowledge to automatically manufacture an end product given a particular requirements specification. Such configuration knowledge models feature interactions either in the problem domain (at the requirements level) or in the solution domain (at the implementation level). Thus, feature interactions are defined as a composition problem in one specific phase of the generative programming lifecycle. However, we experienced the need to model and handle feature interactions that cross the problem and solution domain. This paper presents a specific case study, in the context of our work on distributed runtime adaptation, motivating this important but often ignored category of problem-solution feature interactions.

Keywords. Generative programming, configuration knowledge, feature interactions, distributed runtime adaptation, DyReS

1. Introduction

Generative programming (GP) is a software engineering paradigm based on modelling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge [8]. Typically, this configuration knowledge includes feature conflicts (illegal feature combinations) and feature dependencies (atomic feature groups) at a requirements or implementation level. Such feature interactions are perfectly covered by the more traditional definition of feature interactions where two features that work correctly in isolation (when composed with the base system) do not work correctly any longer when they are combined together and composed with the same base system. As a consequence, the feature interaction problem boils down to a combination problem in one specific phase of the generative programming lifecycle.

This paper presents a case study showing that this traditional view on the feature interaction problem only works for very simple product configuration [21,5] problems with one-to-one mappings between requirements and implementation components. In the case study, we have applied the generative programming approach to the design of distributed runtime adaptations. In this context, we have found very complex configuration
problems which we could not express by means of pairwise feature interactions limited to one GP lifecycle phase. Instead, we experienced the need to model and handle feature interactions that cross the problem and solution domain.

We define a distributed runtime adaptation as binding or unbinding multiple inter-dependent components [22] to/from a running distributed application in a coordinated way [23]. Consider for instance confidentiality as a distributed component to be bound with a client-server based application. The confidentiality component consists of both an encryption and a decryption subcomponent. In this example, the encryption component is responsible for encrypting outgoing messages at the client node while the complementary decryption component contains the functionality for decrypting incoming messages at the server node. Obviously, the encryption and decryption components are mutually interdependent and therefore, coordination support is needed to ensure these components are bound to the application atomically without bringing the overall system into an inconsistent state. The implementation of such coordination support typically involves the design of an adaptation protocol. In general, an adaptation protocol defines the different reconfiguration and coordination steps for each node to execute in order to perform the distributed adaptation in a coordinated way.

![Figure 1](image)

**Figure 1.** A generative programming approach for the design of adaptation protocols.

The basic idea of applying the generative programming approach to the design of a family of adaptation protocols is depicted in Figure 1. An application typically exhibits a number of application-specific characteristics and desires some specific requirements with respect to how the adaptation protocol should coordinate the adaptation. The software design of the family consists of a generic adaptation protocol, that works in all cases, and a set of optional customization strategies that optimize the generic adaptation protocol tailoring it based on the application-specific characteristics and requirements. We consider the application-specific characteristics and requirements to be requirements-level features and customization strategies to be implementation-level features. Part of our domain analysis resulted thus in two separate feature models [15] that distinguish between requirements-level and implementation-level features respectively. This separation also has been inspired by the distinction between product line variability and software variability, as defined by Metzger et al. [18].

However, specifying the most optimal adaptation protocol that is customized to the application-specific characteristics and requirements is very complex as there are many interactions between the characteristics and requirements of the specific application and the different customization strategies. Therefore, this paper makes the case that adopting
the view of problem-solution feature interactions helps to model and handle this complex configuration problem in a better way.

The rest of this paper is structured as follows. In Section 2, we summarize the essential details on the DyReS framework and its design of a generic adaptation protocol. Next, we discuss all relevant requirements- and implementation-level features in Sections 3 and 4 respectively. Section 5 then makes the case for feature interactions that cross the problem and solution domain. Finally, we conclude the paper in Section 6.

2. DyReS and the design of its generic adaptation protocol

DyReS\(^1\) is a framework that we developed [23,14] based on the vision and principles of the NeCoMan middleware [11,13]. Currently, DyReS extends two industrially-used middlewares (Spring [2] and JBoss [1]) with runtime support for implementing distributed adaptations to a client-server based application in a coordinated and safe way.

Existing adaptation support systems typically offer one generic adaptation protocol for implementing and coordinating distributed runtime adaptations [3,6,17,19,4,9]. DyReS, on the contrary, enables the application developer to specify a customized adaptation protocol that is tailored to application-specific characteristics and requirements (one of its main contributions). We and others [10,23,12] already have shown that a customized adaptation protocol can improve a generic solution in terms of performance and adaptation semantics.

However, specifying the most suited adaptation protocol that is customized to the specific application is very complex as there are many interactions between the characteristics and requirements of the application and the different customization strategies. In addition, the configuration of an adaptation protocol is complicated by key safety properties such as global state consistency and structural integrity [20,19,23] that require coordination and synchronization during the distributed adaptation, but these key safety properties are inherent to the problem of dynamic adaptation.

DyReS\(^1\) enables the developer to implement a distributed adaptation as a set of adaptation scripts, one for each node involved in the distributed adaptation. These adaptation scripts then are interpreted and executed by DyReS. Each adaptation script in DyReS typically contains a sequence of four reconfiguration tasks: installation (installing new components), finishing (drive old components and/or the application to a safe state), activation (unbinding any old and binding any new components) and removal (removing old components). Each reconfiguration task is structured as a sequence of one or more reconfiguration actions that are listed below.

Create vs. remove: loading (without binding) vs. unloading a specified component into the running application.
Interrupt vs. resume: driving an component to a safe state vs. resuming execution.
Impose safe state: detecting when an component reaches a safe state.
Bind vs. unbind: binding vs. unbinding an component to and from the application.
Start vs. stop: staring vs. stopping active components that execute in their own thread of control.

\(^1\)DyReS stands for Dynamic Reconfiguration Support. Source code and contact information are available at http://www.cs.kuleuven.be/~distrinet/projects/DyReS.
In addition, DyReS offers two synchronization actions: **sync_wait** for blocking a re-configuration task until it receives a specified synchronization message and **sync_notify** sending a specified synchronization message. Detailed discussion of these primitives in DyReS already has been described elsewhere [14].

*Generic adaptation protocol* Because of the complexity of preserving safety properties, DyReS offers a generic adaptation protocol. The complete partial ordering that makes up the generic adaptation protocol is shown in Figure 2. Rounded boxes and titled rounded boxes are used to represent the reconfiguration actions and tasks respectively; the arrows between pairs of boxes express a *before* relationship and define a partial ordering.

It is also possible in DyReS to manually configure this generic protocol to optimize it further taking the characteristics and requirements of the specific application into account. Unfortunately, this still is a very complex procedure for reasons mentioned earlier. To address this complex configuration problem, we suggest to exploit the interactions discussed in Section 5 to provide automated support for automatically selecting applicable customization strategies.

### 3. Application-specific characteristics

Figure 3 depicts our requirements-level feature model of the relevant application-specific characteristics and requirements in the context of distributed runtime adaptations to client-server based applications. We structured these properties according to three dimensions: quality of service requirements, the application itself and the components.

#### 3.1. Quality of service requirements

Different quality of service (QoS) requirements can have an effect on the protocol for performing a distributed runtime adaptation. These QoS requirements typically represent non-functional requirements [7]. We distinguish between three quality attributes in our
model. First of all, the adaptation protocol might cause a service disruption, e.g. when the application gets frozen in the course of driving components to a safe state. Inclusion of this feature indicates that a small or high service disruption of the application under adaptation is tolerated. Secondly, graceful service degradation tolerance allows for a situation in which one or more of the required QoS requirements temporarily not are met. E.g. when a distributed runtime adaptation involves an application running on a platform with limited CPU resources, exclusion of the higher CPU consumption tolerated feature can be used to avoid overloading the scarce CPU resources. Finally, an application re-
quiring robustness from the adaptation protocol demands that client requests never will be rejected by the server. Robustness is a relevant quality attribute because existing application support to enforce reliable communication and/or to handle state inconsistencies (cfr. Section 3.2) for instance obviously cannot cover the situation where reliability becomes compromised during the execution of a distributed runtime adaptation.

3.2. Application

Properties of the client-server based application that can influence the adaptation protocol for performing a distributed runtime adaptation are state inconsistency tolerance and the communication protocol. If the application is able to cope with state inconsistencies, e.g. via checkpoints combined with a roll-back mechanism [16], the adaptation protocol does not have to cope with state inconsistencies. W.r.t. the communication between client and server, a number of subfeatures are of importance: its synchronous or asynchronous, best effort or reliable and ordered or unordered character. Note that the latter two subfeatures only concern normal application execution.

3.3. Components

Components that are involved in the adaptation may effect the adaptation protocol in different ways depending on the client or server process (or both) they incorporate and on being on old or new component. In addition to the fact that a component can incorporate a client or server process or both, it also should be indicated if the client or server process executes in its own thread of control (is active). An old component (to be replaced or removed) is either stateless or stateful. The execution state of an old component that is stateful can be externally or internally dependent\(^2\) (or both), persistent or transient and it may be the case that stateful old components already incorporate support for state transfer. If so, this facilitates the adaptation protocol. Finally, an explicit compatibility relationship should be expressed if the adaptation concerns a new component that will replace an old component, e.g. indicating that a new component is equivalent with the old version.

4. Customization strategies

The customization strategies depicted in the implementation-level feature model in Figure 4 cover the key design decisions that will shape the concrete adaptation protocol.

1. Adaptation type. This first set of customization strategies depend on the type of distributed adaptation DyReS is conducting. E.g. in the case of a component addition, DyReS safely can omit the execution of all unbind, stop and remove actions since no old components need to be removed.

\(^2\)A component belonging to a distributed service on top of an application entails externally dependent state if this state is shared with the application. If the state only relates to the distributed service, these old components are stateless w.r.t. the rest of the application. More formally, assume that the dependency set of a component C is defined as the union of the set of components that are state-dependent on C and the set of components C is state-dependent on, and the adaptation set of a component C is defined as the set of components that are involved in the adaptation. Than, the state of a component C is externally dependent iff dependency_set(C) \(\supset\) adaptation_set(C), C’s state is internally dependent when dependency_set(C) \(\subseteq\) adaptation_set(C).
2. **Order between finish and activate.** DyReS’s generic adaptation protocol finishes any old components before activating new ones (called *finish before activate*). The downside of this generic solution is that service disruption may occur due to driving involved components to a quiescent or frozen state. By already taking new components into use before old ones are finished, we can benefit from a reduced service disruption. Note that this *activate before finish* strategy involves more than simply swapping the order in which DyReS executes the finishing and activation actions respectively [23].

3. **Activation.** In general, DyReS’s generic adaptation protocol offers distributed synchronization that is needed to correctly activate a new distributed component in a *coordinated* way. Activating a new component consists of resuming execution and potentially starting any internal threads of control. Normally, DyReS only can initiate the execution of the resume action when all necessary components are bound and threads have been started (cfr. Figure 2). However, this coordination can be omitted for particular distributed adaptations (*independent*).

4. **Finishing.** Distributed finishing denotes that imposing a safe state needs to happen in a synchronized, coordinated way. While DyReS’s generic algorithm offers distributed finishing by default, this customization strategy can allow to cut out at least some (*local*) or all (*no finishing*) of the reconfiguration actions (for old components) of the finishing task: interrupt, impose_safe_state and stop. Note that DyReS offers different strategies for both interrupting and imposing a safe state over a component.

5. **State synchronization.** State synchronization is another customization strategy with rather far reaching consequences, but sometimes necessary. E.g. driving components to a safe state might be impossible if not all nodes can be monitored. The most intuitive realizations of state synchronization are state transfer and process failure and communication recovery, but alternatives exist.

6. **Invocation marking and additional message ordering support needed.** The two remaining optional customization strategies can be used to include support for invocation marking and/or add additional message ordering support. Invocation marking allows DyReS to distinguish between old and new components, or compatible components running in parallel. Invocation marking usually depends on some form of dispatching support too.
5. Problem-solution feature interactions as configuration knowledge

In this section, we show how we model the configuration knowledge for generating optimized adaptation protocols from a requirements specification. Furthermore, we aim to illustrate the relevance of specifying problem-solution feature interactions for precisely specifying this configuration knowledge. The configuration knowledge we specify serves two main goals. First, the implementation-level feature model defines theoretically possible combinations of customization strategies. However, a lot of these combinations are inconsistent due to feature conflicts and dependencies. The configuration knowledge exactly deals with this problem by only allowing consistent combinations of implementation-level features. Secondly, the configuration knowledge enables generating custom-made adaptation protocols based on application-specific requirements. We now first give a high-level overview of the configuration knowledge specified and explain next which limitations we experienced with the traditional view of feature interactions being isolated in either the problem or solution domain.

Overview  Table 1 shows the interactions that we found between requirements- and implementation-level features. Note that a requires relationship expresses a dependency while an excludes relationship is used to describe a conflict. We discuss four groups of related interactions (cfr. emphasized text below). Interaction (1), for example, states that if the application requires robustness from the adaptation protocol, only adaptation protocols that finish any old components before new ones are activated can be part of the resulting adaptation protocol as end product. However, a potential performance problem with adaptation protocols customized by the finish before activate strategy is that a temporary service disruption is introduced. If, on the contrary, we can switch the order of some activation and finishing actions, a large part of the service disruption is resolved. A DyReS adaptation protocol automatically can be configured following the activate before finish design decision based on interaction (4), (5) and (6). The latter two are examples of an interaction that cannot be expressed as a binary conflict or dependency since only a combination of multiple features interacts with another feature. Note that both externally dependent or persistent component state exclude both state transfer and this activate before finish customization strategy (cfr. interaction (8)). Important alternative customization strategies to realize less or no service disruption are no or local finishing.

Precise specification of configuration knowledge  During the process of specifying the above configuration knowledge, we encountered the following important observations. First of all, we experienced the need to model feature interactions that cross the problem and solution domain. Otherwise, we were not able to specify the configuration knowledge precisely enough so that it accommodates the two above goals of consistency and application-specific customization. For example, specifying interaction (3) as a dependency between implementation-level features only (i.e., finish before activate requires state synchronization) is not precisely enough. More specifically, this dependency is
Table 1. Feature interactions as configuration knowledge.

<table>
<thead>
<tr>
<th>Feature interaction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) robustness</td>
<td>requires finish before activation.</td>
</tr>
<tr>
<td>(2) incompatible</td>
<td>excludes state synchronization.</td>
</tr>
<tr>
<td>(3) finish before activate</td>
<td>∧ persistent requires state synchronization.</td>
</tr>
<tr>
<td>(4) transient service disruption tolerated</td>
<td>requires activate before finish.</td>
</tr>
<tr>
<td>(5) activate before finish</td>
<td>requires ¬ externally dependent.</td>
</tr>
<tr>
<td>(6) transient externally dependent</td>
<td>∧ state transfer support requires activate before finish.</td>
</tr>
<tr>
<td>(7) activate before finish</td>
<td>requires invocation marking.</td>
</tr>
<tr>
<td>(8) externally dependent</td>
<td>∨ persistent excludes activate before finish ∧ state transfer.</td>
</tr>
<tr>
<td>(9) activate before finish</td>
<td>∧ ordered requires additional message ordering support.</td>
</tr>
<tr>
<td>(10) stateful</td>
<td>activate before finish requires state synchronization.</td>
</tr>
<tr>
<td>(11) transient</td>
<td>∧ externally dependent ∧ activate before finish requires state simulation.</td>
</tr>
<tr>
<td>(12) (persistent</td>
<td>∨ externally dependent) ∧ activate before finish requires state simulation ∧ temporary service degradation tolerated.</td>
</tr>
<tr>
<td>(13) activate before finish</td>
<td>excludes robustness.</td>
</tr>
<tr>
<td>(14) ¬ higher memory footprint tolerated</td>
<td>∧ ¬ higher CPU consumption tolerated excludes invocation marking.</td>
</tr>
<tr>
<td>(15) synchronous</td>
<td>excludes invocation queuing.</td>
</tr>
<tr>
<td>(16) state inconsistency tolerance</td>
<td>∧ temporary service degradation tolerated requires no finishing.</td>
</tr>
<tr>
<td>(17) best effort</td>
<td>∧ equivalent ∧ state inconsistency tolerance requires no finishing.</td>
</tr>
<tr>
<td>(18) synchronous</td>
<td>requires local ∧ thread blocking.</td>
</tr>
<tr>
<td>(19) asynchronous</td>
<td>requires distributed ∧ invocation queuing.</td>
</tr>
<tr>
<td>(20) equivalent</td>
<td>∨ (reliable ∧ state inconsistency tolerance) requires independent.</td>
</tr>
<tr>
<td>(21) ¬ transient service degradation tolerated</td>
<td>excludes independent.</td>
</tr>
<tr>
<td>(22) independent</td>
<td>∧ reliable requires process failure and communication recovery.</td>
</tr>
<tr>
<td>(23) independent</td>
<td>∧ stateful requires state synchronization ∧ no finishing.</td>
</tr>
<tr>
<td>(24) finishing</td>
<td>∧ synchronous ∧ ¬ (active client ∨ active server) requires local.</td>
</tr>
<tr>
<td>(25) independent</td>
<td>∧ ¬ equivalent requires state synchronization ∧ ¬ state transfer.</td>
</tr>
<tr>
<td>(26) state synchronization</td>
<td>∧ ¬ state transfer requires temporary service degradation tolerated.</td>
</tr>
<tr>
<td>(27) independent</td>
<td>∧ stateless ∧ ¬ addition only requires activate before finish.</td>
</tr>
</tbody>
</table>

only true if the application has persistent state. Indeed, an application without persistent state would benefit from an optimized adaptation protocol that skips state synchronization. Similarly, interactions between requirements-level features can also depend on implementation-level features (e.g. in (22)). Secondly, problem-solution feature interactions cannot be expressed as binary relations, but in fact are N-ary relations. Most of the interactions are triple relations but some, such as interaction (12), relate up to 5 features.

6. Conclusion

We have motivated the case for feature interactions that cross problem and solution domain in order to model complex configuration knowledge. In particular, we presented our initial results of applying the generative programming approach to the design of adaptation protocols. We use a requirements-level feature model to represent application-specific characteristics and requirements while an implementation-level feature model
represents customization strategies that can be applied to a generic adaptation protocol. We have shown that the configuration knowledge for generating a correct adaptation protocol from a requirements specification is complex. In particular, interactions between implementation-level features can only be precisely specified by taking into account requirements-level features, and vice versa. As a result, these problem-solution interactions are generally N-ary relations.

References