From Goals to High-Variability Software Design

Yijun Yu[1⋆], Alexei Lapouchnian[2⋆⋆], Sotirios Liaskos[3⋆⋆⋆], John Mylopoulos[2†], and Julio C.S.P. Leite[4‡]

[1] Department of Computing, The Open University, United Kingdom
[2] Department of Computer Science, University of Toronto, Canada
[3] School of IT, York University, Canada
[4] Department of Informatics, PUC-Rio, Brazil

Abstract. Software requirements consist of functionalities and qualities to be accommodated during design. Through goal-oriented requirements engineering, stakeholder goals are refined into a space of alternative functionalities. We adopt this framework and propose a decision-making process to generate a generic software design that can accommodate the full space of alternatives each of which can fulfill stakeholder goals. Specifically, we present a process for generating complementary design views from a goal model with high variability in configurations, behavioral specifications, architectures and business processes.

1 Introduction

Traditionally, requirements consist of functions and qualities the system-to-be should support [8, 4]. In goal-oriented approaches [26, 27, 22], requirements are derived from the list of stakeholder goals to be fulfilled by the system-to-be, and the list of quality criteria for selecting a solution to fulfill the goals [22]. In goal models, root-level goals model stakeholder intentions, while leaf-level goals model functional system requirements. [26] offers a nice overview of Goal-Oriented Requirements Engineering, while the KAOS [8] and the i*/Tropos approaches [30, 2] represent the state-of-the-art for research on the topic.

We are interested in using goal models to develop *generic* software solutions that can accommodate many/all possible functionalities that fulfill stakeholder goals. This is possible because our goals models are extensions of AND/OR graphs, with OR decompositions introducing alternatives into the model. The space of alternatives defined by a goal model can be used as a basis for designing fine-grained variability for highly customizable or adaptable software. Through customizations, particular alternatives can be selected by using *softgoals* as criteria. Softgoals model stakeholder preferences, and may represent qualities that lead to non-functional requirements.

⋆ y.yu@open.ac.uk
⋆⋆ alexei@cs.toronto.edu
⋆⋆⋆ liaskos@yorku.ca
† jm@cs.toronto.edu
‡ julio@inf.puc-rio.br

The main objective of this paper is to propose a process that generates a high variability software design from a goal model. The process we propose is supported by heuristic rules that can guide the design.

Our approach to the problem is to accommodate the variability discovered in the problem space by a variability model in the solution space. To this end, we employ three complementary design views: a feature model, a statechart and a component model. The feature model describes the system-to-be as a combination variable set of features. The statechart provides a view of the behavior alternatives in the system. Finally, the component model reveals the view of alternatives as variable structural bindings of the software components.

The goal model is used as the logical view at the requirements stage, similar to the global view in the 4+1 views [17] of the Rational Unified Process. This goal model transcends and circumscribes design views. On the other hand, a goal model is missing useful information that will guide decisions regarding the structure and behavior of the system-to-be. Our proposed process supports lightweight annotations for goal models, through which the designer can introduce some of this missing information.

The remainder of the paper is organized as follows: Section 2 introduces requirements goal models where variability is initially captured. Section 3 talks about generating high-variability design views in feature models, statecharts, component-connector architectures and business processes. Section 4 discusses tool support and maintenance of traceability. Section 5 explains a case study. Finally, Section 6 presents related work and summarizes the contributions of the paper.

2 Variability in Requirements Goal Models

We adopt the formal goal modeling framework proposed in [9, 23]. According to this framework, a goal model consists of one or more root goals, representing stakeholder objectives. Each of these is AND/OR decomposed into subgoals to form a forest. In addition, a goal model includes zero or more softgoals that represent stakeholder preferences. These can also be AND/OR decomposed. Moreover, there can be positive and negative contribution relationships from a goal/softgoal to a softgoal indicating that fulfillment of one goal/softgoal leads to partial fulfillment or denial of another softgoal. The semantics of AND/OR decompositions is adopted from AI planning. [9] and [23] provide a formal semantics for different goal/softgoal relationships and propose reasoning algorithms which make it possible to check (a) if root goals are (partially) satisfied/denied assuming that some leaf-level goals are (partially) satisfied/denied; (b) search for minimal sets of leaf goals which (partially) satisfy/deny all root goals/softgoals.

Figure 1a shows an example goal model describing the requirements for "schedule a meeting". The goal is AND/OR decomposed repeatedly into leaf-level goals. An OR-decomposition of a goal introduces a *variation point*, which defines alternative ways of fulfilling the goal.

Variability is defined as all possible combinations of the choices in the variation points. For example, the four variation points of the goal model in Figure 1a are marked VP1-VP4. VP1 contributes two alternatives, VP2 and VP4 combined contribute 3, while

VP3 contributes 2. Then, the total space of alternatives for this goal model includes 2*3*2 = 12 solutions. Accordingly, we would like to have a systematic process for producing a generic design that can potentially accommodate all 12 solutions.

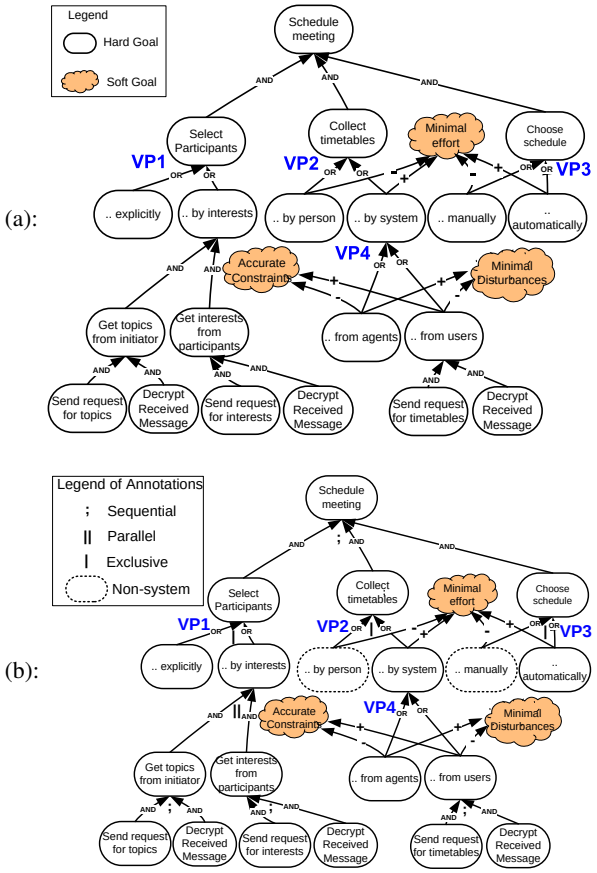

Fig. 1. An example generic goal model of the meeting scheduler: (a) Variation points by OR decompositions are indicated as VP1-4; (b) the goal model annotated with enriched labels

It is desirable to preserve requirements variability in the design that follows. We call a design parameterized with the variation point choices a *generic* design, which can implement a product-line, an adaptive product and/or a component-based architecture. A *product-line* can be configured into various products that share/reuse the implementation of their commonality. An *adaptive* product can adapt its behavior at run-time to accommodate anticipated changes in the environment as it is armed with alternative solutions. A *component-based* design makes use of interface-binding to flexibly orchestrate components to fulfill the goal. In the next section, we will detail these target design views as feature models, statecharts and component-connectors.

In this work traceability between requirements and design is achieved by an explicit transformation from goal models to design views. Goal models elicited from re-

quirements are more abstract than any of the design views; therefore such generation is only possible after identifying a mapping between goals and design elements. To save designer's effort, we enrich the goal models with minimal information such that the generative patterns are sufficient to create preliminary design-level views.

Figure 1b shows an annotated goal model for feature model and statecharts generation, where the semantics of the goal decompositions are further analyzed: (1) VP1, VP2 and VP3 are exclusive (|) and VP4 is inclusive; (2) based on the temporal relationships of the subgoals as required by the data/control dependencies, AND-decompositions are annotated as sequential (;) or parallel (||) and (3) goals to be delegated to non-system agents are also indicated. We will explain the detailed annotations for deriving a component-connector view in the next section. As the preliminary design evolves, design elements and their relationships can change dramatically. However, the traceability to the required variability must be maintained so that one can navigate between the generated design views using the derived traceability links among them.

3 Generating preliminary design views

This section presents three preliminary design views that can be generated from a high-variability goal model. For each view, we use generative patterns to simplify the task.

3.1 Feature models

Feature modeling [14] is a domain analysis technique that is part of an encompassing process for developing software for reuse (referred to as Domain Engineering [7]). As such, it can directly help in generating domain-oriented architectures such as product-line families [15].

There are four main types of features in feature modeling: *Mandatory*, *Optional*, *Alternative*, and *OR* features [7]. A Mandatory feature must be included in every member of a product line family provided that its parent feature is included; an Optional feature may be included if its parent is included; exactly one feature from a set of Alternative features must be included if a parent of the set is included; any non-empty subset of an OR-feature set can be included if a parent feature is included.

There are fundamental differences between goals and features. Goals represent stakeholder intentions, which are manifestations of intent which may or may not be realized. Goal models are thus a space of intentions which may or may not be fulfilled. Features, on the other hand, represent properties of concepts or artifacts [7]. Goals will use the services of the system-to-be as well as those of external actors for their fulfillment. Features in product families represent *system* functions or properties. Goals may be partially fulfilled in a qualitative or quantitative sense [9], while features are either elements of an allowable configuration or they are not. Goals come with a modality: achieve, maintain, avoid, cease [8], while features have none. Likewise, AND decomposition of goals may introduce temporal constraints (e.g., fulfill subgoal A before subgoal B) while features do not.

As noted in [7], feature models must include the semantic description and the rationale for each feature (why it is in the model). Also, variable (OR/Optional/Alternative)

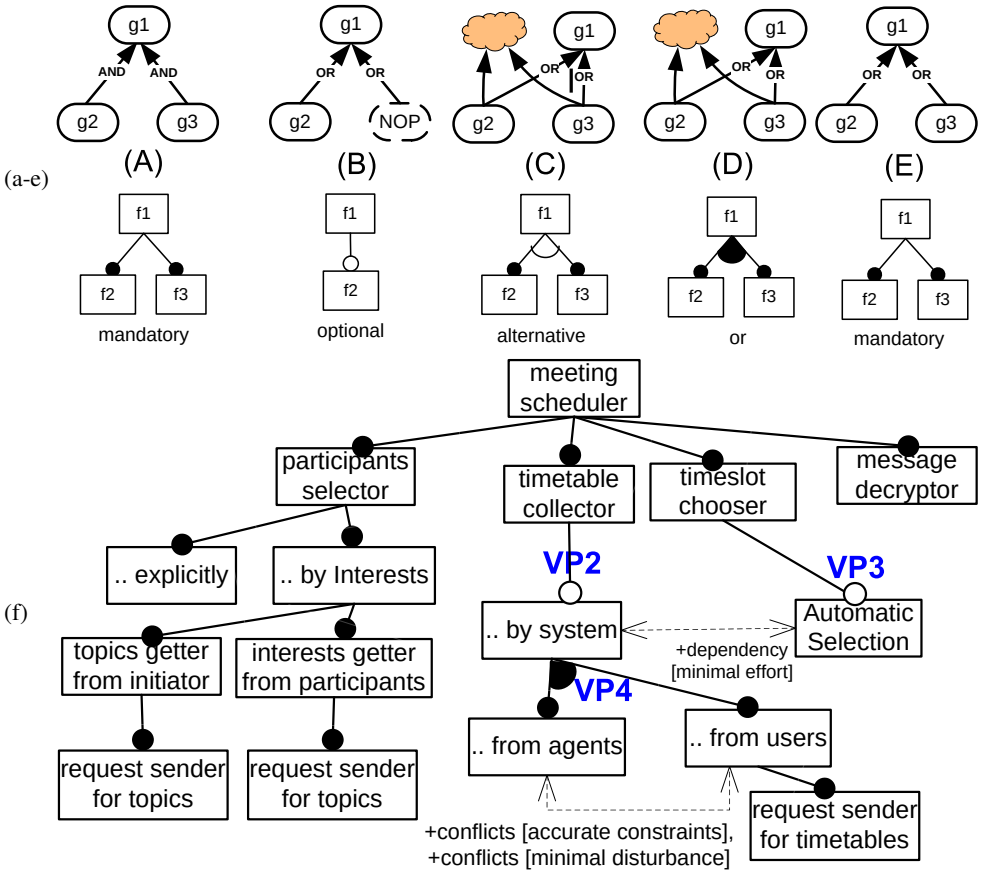

Fig. 2. Generating Features: (a-e) a set of patterns; (f) the feature model generated from Figure 1b

features should be annotated with conditions describing when to select them. Since goal models already capture the rationale (stakeholder goals) and the quality criteria driving the selection of alternatives, we posit that they are proper candidates for the generation of feature models.

Feature models represent the variability in the system-to-be. Therefore, in order to generate them, we need to identify the subset of a goal model that is intended for the system-to-be. Annotations can be applied to generate the corresponding preliminary feature model. AND decompositions of goals generally correspond to sets of Mandatory features (see Figure 2a). For OR decompositions, it is important to distinguish two types of variability in goal models: *design-time* and *runtime*. Design-time variability is high-level and independent of input. It can be usually be bound at design-time with the help of user preferences or other quality criteria. On the other hand, runtime variability depends on the runtime input and must be preserved at runtime. For example, meeting participants can be selected explicitly (by name) or chosen by matching the topic of the meeting with their interests (see Figure 2). The choice will depend on the meeting type and thus both alternatives must be implemented. When subgoals cannot be selected

based on some quality criteria (softgoals), they are considered runtime variability, thus, runtime variability in goal models will generate mandatory features (Figure 2e). Otherwise, as design-time variability, other OR decompositions can be mapped into sets of OR-features (Figure 2d). However, Alternative and Optional feature sets do not have counterparts in the AND/OR goal models. So, in order to generate these types of features we need to analyze whether some of the OR decompositions are, in fact, XOR decompositions (where exactly one subgoal must be achieved) and then annotate these decompositions with the symbol "|" (Figure 2c). The inclusive OR decomposition corresponds to a feature refined into a set of OR features (Figure 2d). Finally, when a goal is OR-decomposed into at least one non-system subgoal (specified by a goal annotation NOP), the sibling system subgoals will be mapped into optional features (Figure 2b). As a result, Figure 2f shows the feature model generated from Figure 1b. The implemented procedure has been reported in [31].

In a more complex design, the system may need to facilitate the environmental actors in achieving their goals or monitor the achievement of these goals. Here, the goals delegated to the environment can be replaced with user interfaces, monitoring or other appropriate features. In general, there is no one-to-one correspondence between goals delegated to the system and features. While high-level goals may be mapped directly into grouping features in an initial feature model, a leaf-level goal may be mapped into a single feature or multiple features, and several leaf goals may be mapped into a feature, by means of factoring. For example, a number of goals requiring the decryption of received messages in a secure meeting scheduling system may be mapped into a single feature "Message Decryptor" (see Figure 2[5]).

3.2 Statecharts

Statecharts, as proposed by David Harel [11], are a visual formalism for describing the behavior of complex systems. On top of states and transitions of a state machine, a statechart introduces nested super-/sub-state structure for abstraction (from a state to its super-state) or decomposition (from a state to its substates). In addition, a state can be decomposed into a set of *AND* states (visually separated by swim-lanes) or a set of *XOR* states [11]. A transition can also be decomposed into transitions among the substates.

This hierarchical notation allows the description of a system's behavior at different levels of abstraction. This property of statecharts makes them much more concise and usable than, for example, plain state machines. Thus, they constitute a popular choice for representing the behavioral view of a system.

Figure 3a0 shows a mapping from a goal in a requirements goal model to a state in a statechart. There are four basic patterns of goals in linear temporal logic formula, here we show mappings for *achieve* and *maintain* goals, whereas *cease* and *avoid* goals are similar.

An *achieve* goal is expressed as a temporal formula with P being its precondition, and Q being its post-condition. In the corresponding statechart, one entry state and one

[5] One can systematically derive feature names from the hard goal descriptions by, for example, changing the action verb into the corresponding noun (e.g., "schedule meeting" becomes "meeting scheduler").

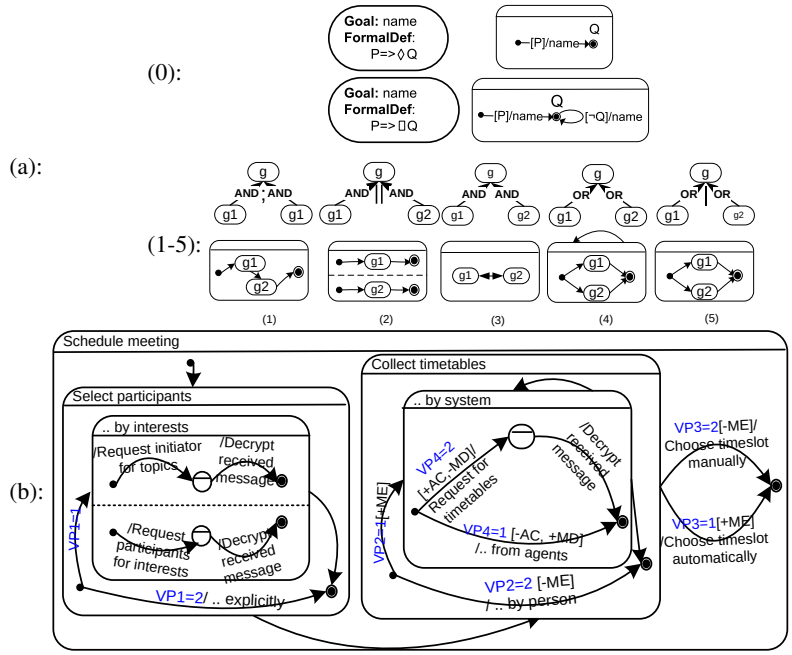

Fig. 3. Generating Statecharts: (a0-a5) a set of patterns; (b) the generated statecharts from Figure 1b

exit state are created: P describes the condition triggering the transition from the entry to the exit state; Q prescribes the condition that must be satisfied at the exit state. The transition is associated with an activity to reach the goal's desired state. The *cease* goal is mapped to a similar statechart (not shown) by replacing the condition at the exit state with $\neg Q$. For mapping a *maintain* goal into a statechart, the transition restores the state back to the one that satisfies Q whenever it is violated while P is satisfied. Similar to the *maintain* goal's statechart, the statechart for an *avoid* goal swaps Q with its negation. These conditions can be used symbolically to generate an initial statechart view, i.e., they do not need to be explicit temporal logic predicates. At the detailed design stage, the designer may provide solution-specific information to specify the predicates for a simulation or an execution of the refined statechart model.

Then applying the patterns in Figure 3a1-5, a goal hierarchy will be mapped into an isomorphic state hierarchy in a statechart. That is, the state corresponding to a goal becomes a super-state of the states associated with its subgoals. The runtime variability will be preserved in the statecharts as alternative transition paths.

The transformation from a goal model to an initial statechart can be automated even when the temporal formulae are not given: we first associate each leaf goal with a state that contains an entry substate and an exit substate. A default transition from the entry substate to the exit substate is labeled with the action corresponding to the leaf goal. Then, the AND/OR goal decompositions are mapped into compositions of the statecharts. In order to know how to connect the substates generated from the corresponding AND-decomposed subgoals, temporal constraints are introduced as goal model anno-

tations, e.g., for an OR-decomposition, one has to consider whether it is inclusive or exclusive.

Given root goals, our statechart generation procedure descends along the goal refinement hierarchy recursively. For each leaf goal, a state is created according to Figure 3a0. The created state has an entry and an exit substates. Next, annotations that represent the temporal constraints with the AND/OR goal decompositions are considered. Composition patterns can then be used to combine the statecharts of subgoals into one statechart (Figure 3a1-5). The upper bound of number of states generated from the above patterns is $2N$ where N is the number of goals.

For the Schedule Meeting goal model in Figure 1, we first identify the sequential/parallel control patterns for AND-decompositions through an analysis of the data dependencies. For example, there is a data dependency from "Send Request for Timetable" to "Decrypt Received Message" because the time table needs to be requested first, then received and decrypted. Secondly, we identify the inclusive/exclusive patterns for the OR decompositions. For example, "Choose Time Slot" is done either "Manually" or "Automatically". Then we add transitions according to the patterns in Figure 3a. As a result, we obtain a statechart with hierarchical state decompositions (see Figure 3b). It describes an initial behavioral view of the system.

The preliminary statechart can be further modified by the designer. For example, the abstract "send requests for timetable" state can be further decomposed into a set of substates such as "send individual request for timetable" for each of the participants. Since the variability is kept by the guard conditions on the transitions, changes of the statecharts can still be traced back to the corresponding annotated goal models. Moreover, these transitions specified with entry/exit states can be used to derive test cases for the design.

3.3 Component-connector view

A component-connector architectural view is typically formalized via an architecture description language (ADL) [21]. We adapt the Darwin ADL [20] with elements borrowed from one of its extensions, namely Koala [28]. Our component-connector view is defined by components and their bindings through interface types. An *interface type* is a collection of message signatures by which a component can interact with its environment. A *component* can be connected to other components through instances of interface types (i.e. interfaces). A **provides** interface shows how the environment can access the component's functionality, whereas a **requires** interface shows how the component can access the functionality provided by the environment. In a component configuration, a **requires** interface of a component in the system must be *bound* to exactly one **provides** interface of another component. However, as in [28], we allow alternative bindings of interfaces through the use of a special connection component, the *switch*. A switch allows the association of one **requires** interface with many alternative **provides** interfaces of the same type, and is placed within a compound component which contains the corresponding alternative components.

A preliminary component-connector view can be generated from a goal model by creating an interface type and a component for each goal. The interface type contains the signature of an operation. The operation name is directly derived from the goal

description, the **IN/OUT** parameters of the operation signature must be specified for the input and output of the component. Therefore in order to generate such a preliminary component-connector view, each goal needs to be annotated with specification of input/output data. For example, the interface type generated from the goal "Collect Timetables from Users" is shown as follows:

```
interface type ICollectTimetablesFromUsers {
   CollectTimetables(IN Users, Interval,
                     OUT Constraints);
}
```

The generated component implements the interface type through a **provides** interface.

The **requires** interfaces of the component depend on how the goal is decomposed. If the goal is AND-decomposed, the component has as many **requires** interfaces as the subgoals. In our example, the initial component of the goal "Collect timetables from Users" is generated as follows:

```
component TimetableCollectorFromUsers {
      provides ICollectTimetables;
      requires IGetTimetable, IDecryptMessage;
}
```

The **requires** interfaces are bound to the **provides** interfaces of the subgoals.

If the goal is OR-decomposed, the interface types of the subgoals are first replaced with the interface type of the parent goal such that the **provides** interface of the parent goal is of the same type as the **provides** interfaces of the subgoals. Then, inside the generated component, a switch is introduced to bind these interfaces. The **provides** interface of the compound component generated for the parent goal can be bound to any of the subgoal's **provides** interfaces. Both the switch and the components of the subgoals are placed inside the component of the parent goal, and are *hidden* behind its interface.

In Figure 4, the graphical notation is directly adopted from Koala/Darwin. The boxes are components and the arrows attached to them represent **provides** and **requires** interfaces, depending on whether the arrow points inwards or outwards respectively. The lines show how interfaces are bound for the particular configuration and are annotated with the name of the respective interface type; the shape of the overlapping parallelograms represents a switch. Patterns show how AND/OR decompositions of system goals are mapped into the component-connector architecture.

In order to accommodate a scheme for *event* propagation among components, we follow an approach inspired by the C2 architectural style [21] where requests and notifications are propagated in opposite directions. As requests flow from high level components to low-level ones, notifications (events) originated from low-level components will propagate to high-level ones. Such events are generated from components associated with goals that are delegated to the environment (non-system goals). These components are responsible for supporting external actors' activity to attain the associated goal, to sense the progress of this attainment and to communicate it to the components of higher level by generating the appropriate events. We name such components *interface components* to signify that they lay at the border of the system. Interface components have an additional **requires** interface that channels the events. This interface is optionally bound to an additional **provides** interface at the parent component, its event

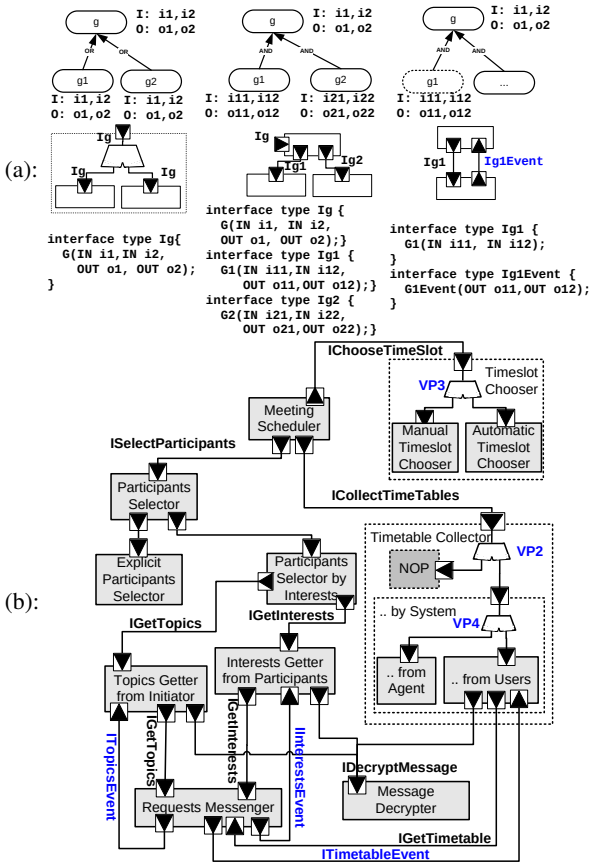

Fig. 4. Generating Architectures: (a) a set of patterns; (b) the generated architecture from Figure 1b

handler. In Java, such a binding becomes a Listener interface in the parent component for receiving events from the interface component.

In our example (see Figure 4), three goals "Send request for topics/interests/timetable" are delegated to external actors (e.g. initiator, participants and users), and will therefore yield an interface component, e.g.:

```
component TimetableCollectorFromUsers {
      provides ICollectTimetable, ITimetableEvent;
      requires IGetTimetable, IDecriptMessage;
}
```

The `RequestMessenger` component is a result of merging components of lower level and is being reused by three different components of higher level through parameterization. These are examples of modifications the designer may chose to make, after the preliminary configuration has been produced.

3.4 Business processes view

We have shown how to transform an annotated goal model into traditional detailed views, in terms of configuration (i.e., feature models), behavior (i.e., statecharts) and structure (i.e., component-connector architectures). By similarity in abstraction, one may prefer to generate high-variability design views for service-oriented architectures. In fact, i* goal models, an extension of the presented goal model language, have long been proposed to model business processes engineering for agent-oriented software [30]. Using the concept of actors in i*, one can group together goals belonging to the same stakeholders and refine them from those stakeholders' point of view, also modeling goal delegations among the actors as strategic dependencies. On top of variability in data/control dependencies, we introduced the variability in delegation dependencies. Combining the variability support in data, control and delegation dependencies, we were able to generate high-variability business process models in WS-BPEL [19]. In this work, the interface types of component-connector views in WSDL (Web service definition language) were implemented by Web services and controlled by another Web service. This controlling Web service, implemented in BPEL, ochestrates the constituent services through the patterns similar to what we have defined in the statechart view (i.e., substitute state machines with processes and transitions with flows). In addition, we created another Web service that is capable of configuring BPEL processes based on user preferences such that the high-variability BPEL can be dynamically reconfigured. Since the controlling BPEL process is deployed also as a Web service, one can build atop a hierarchical high-variability design that can be dynamically configured while different layers of goal models are changed. Using the monitoring and diagnosing framework [29] for such changes, we envision this business process model as a promising solution towards a requirements-driven autonomic computing system design where each goal-model generated controller is an element of the managing component in such systems [18].

4 Tool support and traceability

On the basis of the metamodel of annotated high-variability goal models, we developed a model-driven goal modeling tool, OpenOME[6], which is capable of editing goal models, adding annotations, and generating the design views such as feature models [31] and BPEL processes [19]. A simplified goal model in Figure 1 is represented by the XML document in Figure 5a.

Our OpenOME modeling tool is an Eclipse-based graph editors in which the models are persisted in XMI format which is transparent to the modeller. To illustrate the underlying traceability support, here we explain using the XMI examples, in which the default name space is declared by the `xmlns` attribute, informing a goal graph editor that it is a goal model. Next to the annotations to the goals and its refinements, the sibling tags of other namespaces provide traceability links to the other views. The attribute `src` of design elements indicates the origin element in the goal model before the transformation. When the attribute has a value other than `generated`, it indicates that the

[6] http://www.cs.toronto.edu/km/openome

```
<view  xmlns="urn:goals" xmlns:e="urn:enrichments"
  xmlns:f="urn:features" xmlns:s="urn:statecharts"
  xmlns:c="urn:components">
  <goal name="Schedule Meeting">
    <refinement type="goal" value="AND"/>
    <e:refinement type="state" value="SEQ"/>
    <e:annotation type="feature" value="SYS"/>
    <e:annotation type="component"
         input="meeting" output="schedule"/>
    <e:annotation type="state" pre="meeting!=null"
      post="!schedulable OR schedule!=null"/>
    <f:feature name="Meeting Scheduler"
      src="generated: renamed"/>
    <s:statechart name="Schedule Meeting"
      src="generated"/>
    <c:component name="Meeting Scheduler"
      src="generated: renamed"/>
    <goal name="Select Participants"> ... </goal>
    <goal name="Collect Timetables"> ... </goal>
    <goal name="Choose Schedule"> ... </goal>
  </goal>
</view>
```

(a): high-variability goal model annotated

```
<view xmlns:g="urn:goals"  xmlns="urn:features"...>
 <g:goal name="Schedule Meeting">
  <g:refinement type="goal" value="AND"/>...
  <e:annotation type="feature" value="SYS"/>...
  <feature name="meeting scheduler"
   src="renamed"> <refinement type="feature"
    value="AND" src="generated"/>
   <cardinality value="1" src="generated"/>
   <g:goal name="Select Participants">
    <feature name="participants selector"
     src="renamed"> ...</feature>...</g:goal>
    <g:goal name="Collect Timetables">s
     <feature name="timetable collector"
      src="renamed">...</feature>...</g:goal>
    <g:goal name="Choose Schedule">
     <feature name="timetable collector"
      src="renamed">...</feature>...</g:goal>
    <feature name="message decryptor"
     src="refactored"
     origin="Decrypt Received Message">
     ...<feature/> </feature> ...</g:goal> </view>
```

(b): generated features

Fig. 5. Traceability support in representations of annotated goal models

design element has been manually changed. Thus, if the source goal model is changed, the renamed design element will not be overridden by the generation process.

We use the generated feature model view to illustrate the concept of traceability (Figure 5b). The default namespace here is "features" and the nesting structure for feature model was mostly generated from the nesting structure of goals. However, some features (e.g. "message decryptor") can correspond to a goal that is not originally decomposed from that corresponding to its parent. In this case, the original goal with which it was associated is indicated. During design, features not corresponding to any goal can be added/removed. However, such design changes cannot remove existing traceability links including the ones implied by the sibling XML elements. Deleted design elements are still kept in the model with a "deleted" `src` attribute. As the `src` attribute maintains the traceability between a goal and a design element, the link between two design views can be derived. For example, in the above document, one can obtain the traceability link between a "Meeting Scheduler" component and a "meeting scheduler" feature since they are both mapped from the "Schedule Meeting" goal.

An XMI representation of the above metamodel represents the elements of a unified model in a single name space. For each design view, the generation procedure is implemented as an XSLT stylesheet that transforms the unified model into an XML design document that can be imported by the corresponding visual tool. A generated document separates the concerns of various views into different namespaces. The default name space concerns the primary design view, whereas the relationships between adjacent tags of different namespaces capture the traceability among corresponding design views.

5 A case study

A case study was conducted to validate our approach. First we developed a goal model by decomposing the user's goal for *Prepare an Electronic Message* into 48 goals with 10 AND- and 11 OR-decompositions [13]. Instead of developing from scratch we considered reusing components from an existing email client. To support the *external* validity

of our study, we chose a public-domain Java email client Columba[7], which had been studied for reverse engineering of its goal models [34]. This poses a threat to the *internal* validity since none of the authors is involved in the development of Columba, thus our understanding of its absent design alternatives is limited. However, we relied on program understanding to analyze its implemented requirements. Due to the absence of early decisions, the purpose of this study differs from [34] which was to recover as-is requirements from the code. Rather, we compared the implementation with respect to the goal model that appears in [13] in order to check (1) whether we can reuse any components in Columba to implement our generated design views; (2) whether Columba system can cover all the variability needed by our requirements goal model; (3) whether our final design can be traced back to the original requirements goal models.

The study was done as follows (for details see technical report [33]). First, we analyzed the system to reconstruct design views including feature models, component-connector views and statecharts. Starting with the component-connector view: each component corresponds to at least one JAR archive. The system includes 3 basic components (`Core`, `AddressBook`, `Mail`), 23 third party components and 16 plug-in components. All of them become features in the system's feature model view, including 26 mandatory features and 22 optional features. Further, we found that 8 mandatory components are actually related to process-oriented goals for maintainability, such as *Testing* (`junit`, `jcoverage`), *Coding convention check* (`checkstyle`), *Command line options* (`common-cli`), *XML documentations* (`jdom`) and *Build automation* (`ant, maven, jreleaseinfo`). The other 40 components relate to product-oriented user goals. Some AND-decomposed features correspond to OR-decomposed goals. For example, the *Look and Feel* usability goal was implemented by 6 plugin components (`Hippo, Kunststoff, Liquid, Metoulia, Win, Thin`). At run-time, only one of these components will be enabled for the look and feel. Similar observation applies to the `IMAP` and `POP3` server features. Statecharts were obtained from the dynamic messaging trace rather than from static call graphs. Since we are interested in abstract statecharts that are closer to goals, the intra-procedural messages were ignored as long as they were structural and goals could thus be extracted and partially validated by the existing 285 test cases [34].

Since the feature models and the component-connector views were recovered from static call graphs, whereas the statecharts were recovered from dynamic messaging traces, we found it hard to establish traceability among them. For example, a non-functional maintenance goal (e.g. test cases, plugin support) often crosscuts multiple components [32, 34]. In short, there is no one-to-one mapping between goals and the tangled design elements.

Then we looked at reusing Columba to fulfill our high-variability goal, "Prepare an electronic message" [13]. We found that Columba does have reusable components to fulfill some goals of our user. For example, the above hard goal is AND decomposed into *Addressbook* and *Mail folder* management, *Composing email*, *Checking Emails*, *Sending email*, *Spell Checking*, *Printing*, etc. They are implemented by the 3 basic components. The 3rd party components are called by these components to fulfill hard goals such as *Java Mail delivery* (`Ristretto`), *Spam filtering* (`Macchiato`,

[7] http://www.columbamail.org

SpamAssassin), *Full Text search* (Lucene), *Profile Management* (jpim and Berkeley DB je). The plug-ins help with functional goals such as *Chatting* (AlturaMessenger), *mail importing*(Addressbook, mbox, etc.) and *notification* (PlaySoundFilter). For some non-functional softgoals, *Usability* is implemented with 7 *GUI* components (Swing, JGoodies, Frapuccino, jwizz, JDic) and *Online Help* (jhall, usermanual); *Security* is fulfilled by component GNU Privacy guard (JSCF).

There is still missing functionality or variability for our users: *Composing email* does not allow *Compose by voice* subgoal, which we implemented by switching Columba's text-based composer to the speech-based composer using JavaMedia; the mandatory *Auto Completion for Recipients* feature may not satisfy some of our users who wanted *Low Mistake Probability* and *Maximum Performance* when the address book is large. Thus, it was turned into optional in our implementation.

Applying our approach, we obtained preliminary design views that were traceable to each other. The derived feature view consists of 39 features where 9 non-system goals were discarded as NOP. Among the 11 variation points, 2 were turned into mandatory feature decompositions because there was no softgoal associated with them as a selection criterion. The derived statecharts view consists of 21 abstract leaf states where the transitions were turned into test cases. The component-connector view consists of 33 concrete components controlled by 9 switches. We have implemented such generic design based on Columba and additional libraries, among which a reasoning component [9, 23] was used. Thus the resulting system can be reconfigured using quality attributes derived from requirements goals, user skills and preferences [13].

6 Related work and conclusion

There is growing interest on the topic of mapping goal-oriented requirements to software architectures. Brandozzi et al [1] recognized that requirements and design were respectively in problem and solution domains, thereby a mapping between a goal and a component was proposed for increasing reusability. A more recent work by van Lamsweerde et al [25] derives software architectures from the formal specifications of a system goal model using heuristics. Specifically, the heuristics discover design elements such as classes, states and agents directly from the temporal logic formulae that define the goals. Complementary to their formal work, we apply light-weight annotations to the goal model in order to derive design views: if one has the formal specifications for each goal, some heuristics provided in [25] can be used to find the annotations we need, such as system/non-system goals, inputs/outputs and dependencies among the subgoals. Generally, this line of research has not addressed variability at the design level.

Variability within a product-line is another topic receiving considerable attention [5, 7], which has not addressed the problem of linking product family variability to stakeholder goals (and the alternative ways these can be achieved). Closer to our work, [10] proposes an extension of use case notation to allow for variability in the use of the system-to-be. More recently [3], the same group tackled the problem of capturing and characterizing variability across product families that share common requirements.

We maintain traceability between requirements and the generated design using concepts from literate programming and model-driven development. In *literate program-*

ming [16], any form of document and program can be put together in a unified form, which can then be tangled into an executable program with documented comments or into a document with illustrative code segments. In this regards, our annotated goal model is a unified model that can derive the preliminary views with traceability to other views. In *model-driven development* (MDD), code can be generated from design models and a change from the model can propagate into the generated code [24]. To prevent a manual change to the code from being overridden by the code generation, a special *not generated* annotation can be manually added to the comments of the initially generated code. We use MDD in a broader sense – allow designers to change the generated preliminary design views and to manually change the *not generated* attribute of a generated design element. Comparing to deriving traceability links through information retrieval [12], our work proproses a generative process where such links are produced by the process itself and recorded accordingly.

In [6], softgoal models that represent non-functional requirements were associated with the UML design views. On the other hand, this paper focuses on establishing traceability between functional requirements and the preliminary designs. Since variability is explicitly encoded as alternative designs, the non-functional requirements in goal models are kept in the generic design to support larger user bases.

In summary, this paper proposes a systematic process for generating complementary design views from a goal model while preserving variability (i.e., the set of alternative ways stakeholder goals can be fulfilled). The process is supported by heuristic rules and mapping patterns. To illustrate, we report a case study reusing public domain software. The generated preliminary design is comparable in size to the goal model.

References

[1] M. Brandozzi and D. E. Perry. Transforming goal oriented requirements specifications into architectural prescriptions. In *STRAW at ICSE01*, 2001.

[2] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[3] S. Bühne, K. Lauenroth, and K. Pohl. Modelling requirements variability across product lines. In *RE'05*, pages 41–50, 2005.

[4] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishing, 2000.

[5] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Boston, MA, Addison-Wesley, 2001.

[6] L. M. Cysneiros and J. C. S. P. Leite. Non-functional requirements: from elicitation to conceptual models. *IEEE Trans. on Softw. Eng.*, 30(5):328–350, May 2004.

[7] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, MA, USA, June 2000.

[8] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1–2):3–50, Apr. 1993.

[9] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani. Reasoning with goal models. *LNCS*, 2503:167–181, 2002.

[10] G. Halmans and K. Pohl. Communicating the variability of a software-product family to customers. *Software and Systems Modeling*, 2:15–36, 2003.

[11] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Trans. on Software Engineering and Methodology*, 5(4):293–333, Oct. 1996.

[12] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing:the study of methods. *IEEE Trans. on Softw. Eng.*, 32(1):4–19, 2006.

[13] B. Hui, S. Liaskos, and J. Mylopoulos. Goal skills and preference framework. In *International Conference on Requirements Engineering*, 2003.

[14] 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-21, ada235785). Technical report, 1990.

[15] K. C. Kang, S. Kim, J. Lee, and K. Lee. Feature-oriented engineering of PBX software for adaptability and reuseability. *SPE*, 29(10):875–896, 1999.

[16] D. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, 1984.

[17] P. Kruntchen. Architectural blueprints – the "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, Nov. 1995.

[18] A. Lapouchnian, S. Liaskos, J. Mylopoulos, and Y. Yu. Towards requirements-driven autonomic systems design. In *DEAS '05*, pages 1–7, New York, NY, USA, July 2005. ACM Press.

[19] A. Lapouchnian, Y. Yu, and J. Mylopoulos. Requirements-driven design and configuration management of business processes. In *BPM*, pages 246–261, 2007.

[20] J. Magee and J. Kramer. Dynamic structure in software architectures. In *The 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, 1996.

[21] N. Medvidovic and R. N. Taylor. A framework for classify-ing and comparing architecture description languages. *SIGSOFT Softw. Eng. Notes*, 22(6):60–76, 1997.

[22] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Trans. on Softw. Eng.*, 18(6):483–497, 1992.

[23] R. Sebastiani, P. Giorgini, and J. Mylopoulos. Simple and minimum-cost satisfiability for goal models. In *CAiSE*, pages 20–35, 2004.

[24] B. Selic. The pragmatics of model-driven development. *IEEE Softw.*, 20(5):19–25, 2003.

[25] A. van Lamsweerde. From system goals to software architecture. In *Formal Methods for Software Architectures, LNCS 2804*, 2003.

[26] A. van Lamsweerde. Goal-oriented requirements engineering: From system objectives to UML models to precise software specifications. In *ICSE 2003*, pages 744–745, 2003.

[27] A. van Lamsweerde and L. Willemet. Inferring declarative requirements from operational scenarios. *IEEE Trans. Software Engineering*, 24(12):1089–1114, Nov. 1998.

[28] R. C. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, 2000.

[29] Y. Wang, S. A. McIlraith, Y. Yu, and J. Mylopoulos. An automated approach to monitoring and diagnosing requirements. In *ASE*, pages 293–302, 2007.

[30] E. S. K. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *RE'97*, pages 226–235, 1997.

[31] Y. Yu, A. Lapouchnian, J. Leite, and J. Mylopoulos. Configuring features with stakeholder goals. In *ACM SAC RETrack 2008.*, 2008.

[32] Y. Yu, J. Leite, and J. Mylopoulos. From requirements goal models to goal aspects. In *International Conference on Requirements Engineering*, 2004.

[33] Y. Yu, J. Mylopoulos, A. Lapouchnian, S. Liaskos, and J. C. Leite. From stakeholder goals to high-variability software design, ftp.cs.toronto.edu/csrg-technical-reports/509. Technical report, University of Toronto, 2005.

[34] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. C. S. do Prado Leite. Reverse engineering goal models from legacy code. In *RE'05*, pages 363–372, 2005.