From Stakeholder Goals to High-Variability Software Design

Yijun Yu, John Mylopoulos, Alexei Lapouchnian, Sotirios Liaskos
Department of Computer Science, University of Toronto
{yijun,jm,alexei,liaskos}@cs.toronto.edu

Julio C.S.P. Leite
PUC-Rio
{julio@inf.puc-rio.br}

Abstract

Software requirements consist of functionalities and qualities to be accommodated during design. Through goal-oriented requirements engineering, stakeholder goals are analyzed into a model that defines a space of alternative functionalities that can fulfill these goals. We adopt this framework and propose a decision-making process to generate a generic software design that can accommodate the full space of alternative functionalities each of which can fulfill stakeholder goals. Specifically, we present a process for generating three complementary design views from a goal model: a feature model, a set of statecharts and a component/connector architecture. The process is supported by heuristic rules that guide the design. We demonstrate the process through a case study of an open-source email system.

1 Introduction

Traditionally, requirements consist of functions and qualities the system-to-be should support [6, 3]. In goal-oriented approaches [22, 17], 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 fulfil the goals [17]. Goal models have been proposed as vehicles for bridging "early" and "late" requirements [23]. Root-level goals model stakeholder intentions, leaf-level goals model functional system requirements. [22] offers a nice overview of Goal-Oriented Requirements Engineering, while the KAOS approach [6] represents the state-of-the-art for research on the topic.

We are interested in using goal models to generate *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. The space of alternatives defined by a goal model can be used as a basis for designing fine-grain variability for highly customizable software. Customizations can be selected by using *softgoals* as criteria. Softgoals represent stakeholder preferences, and may represent qualities that lead to non-functional requirements.

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. Of course, these rules are only suggestive of how the design is to be moved forward and can be overridden by the designer.

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 prescribes the system-to-be as a variable combination of configurable features. The statechart provides a view of the alternatives in the system behavior. 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 [14] 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 rest of the paper is organized as follows: Section 2 introduces goal models through an example. Section 3 presents the configuration variability view represented by a feature model, while Section 4 presents the behavioral view as generated statecharts. The structural view is presented in Section 5 in terms of component models. In Section 6, we discuss tool support and a case study undertaken to validate the proposed process. Finally, Section 7 presents related work and summarizes the contributions of the paper.

2 Requirements Expressed in Goal Models

We adopt the formal goal modeling framework proposed in [8, 20]. According to this framework, a goal model con-

sists 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. [8] and [20] 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 1 shows a goal model where the root goal is "schedule meeting", while softgoals include "minimal effort" (to schedule a meeting) and "minimal disturbance". Each goal is AND/OR decomposed repeatedly into leaf-level goals such as "send request for topics" and "decrypt received message". These goals are assumed to have corresponding actions that an (external) actor or the system itself can perform to fulfill them. OR decompositions introduce *variation points* which lead to alternative ways of fulfilling higher-level goals, for example, the four variation points of the goal model in Figure 1 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'd like to have a systematic process for producing a generic design that can accommodate all 12 solutions.

Since a generic goal model is a highly abstract description of the early requirements, it needs to be annotated with extra information in order to facilitate design decisions (see patterns in Figure 3, 6, 8). Figure 2 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, AND decompositions are annotated as sequential (;) or parallel (||) and (3) non-system goals (NOP) are also indicated by dotted shapes. We will explain the detailed annotations for deriving a component-connector architecture in Section 5.

To deliver a generic design, the alternatives must be preserved in different design views such as *configuration variability* in feature models, *behavioral variability* in statecharts, and *structural variability* in components. The designer can make detailed changes to these views. However, they must maintain the variability as needed by the stakeholders.

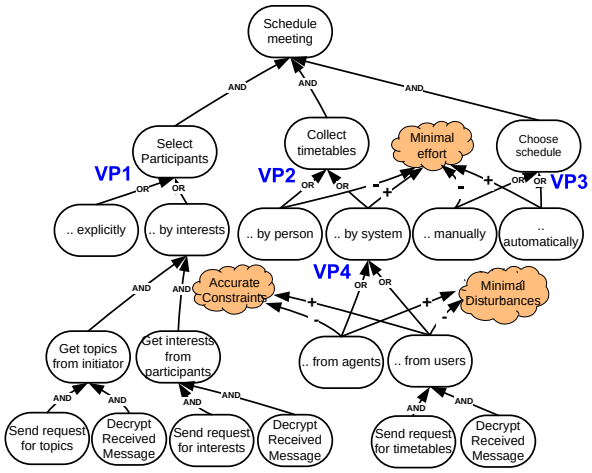

Figure 1. An example generic goal model of the meeting scheduler. Variation points by OR decompositions are indicated as VP1-4.

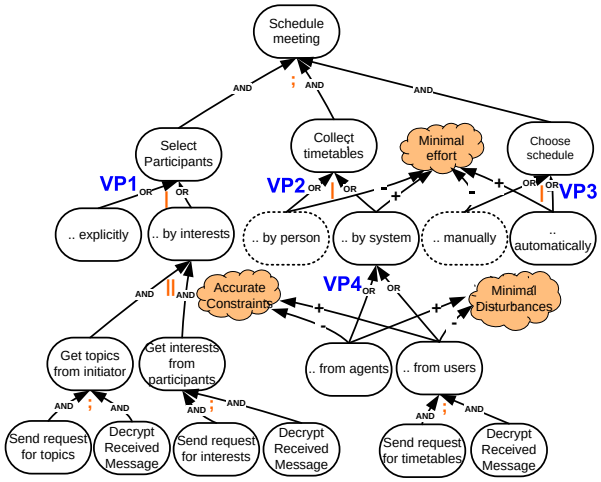

Figure 2. An annotated goal model

In the following three sections, we discuss the generation of the three above-mentioned views from goal models. For each view, we first describe its notation and explain why it is of interest to us. Then we analyze what is the minimal information needed for generate that view from the generic goal model. Finally, we illustrate a process of generating the view from an enriched goal model, using the same example throughout the paper.

3 Generating feature models

The systematic discovery and exploitation of commonality across related software systems is a fundamental technical requirement for achieving successful software reuse [19]. Thus, the software reuse community has long

been interested in analyzing commonality among closely related software systems (called a *product line* or a *domain*). A method called Feature-Oriented Domain Analysis [12] was the first to use *features* for analyzing and representing commonality and variability among applications in a domain. There, members of a software product line are characterized by their features, so variability in a product line can be represented by a feature model.

Feature modeling is a domain analysis technique that is part of an encompassing process for developing software for reuse (referred to as Domain Engineering [5]). As such, it can directly help in generating domain-oriented architectures and software components [13].

There are four main types of features in feature modeling: *Mandatory*, *Optional*, *Alternative*, and *OR* features [5]. 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. Features can also be distinguished based on their mapping into software components [5]. There can be *concrete* features that may be realized as single components, *aspectual* features that affect a number of components, *abstract* features such as performance, and so on, and *grouping* features that may either correspond to a common component interface or be used for organizational purposes.

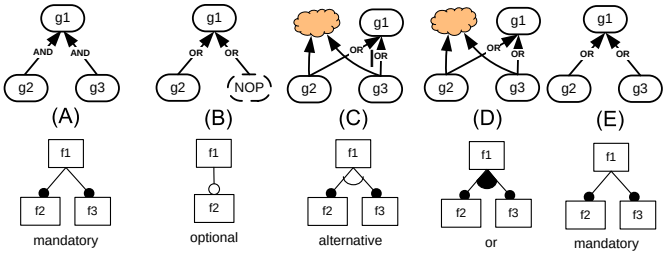

Figure 3. Patterns for generating feature models.

There are fundamental differences between goals and features. Goals represent stakeholder intentions and can be much-removed from any software system-to-be (e.g., "increase production"). Features, on the other hand, represent properties of concepts or artifacts [5]. 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 [8], while features are either elements of an allowable configuration or they are not. Goals come with a modality: achieve, maintain, avoid, cease [6], 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 [5], feature models must include the semantic description and the rationale for each feature (why it is in the model). Also, variable (OR/Optional/Alternative) features should be annotated with conditions describing when to select them. Goal models already capture the rationale (stakeholder goals) and the quality criteria driving the selection of alternatives. Thus, they can be used for generating feature models.

Feature models represent the variability in the system-to-be. Therefore, in order to generate feature models, we need to identify the subset of a goal model that is intended for the system-to-be. Further annotations can be applied to generate the corresponding preliminary feature model. AND decompositions of goals generally correspond to sets of Mandatory features (see Figure 3a). 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 bound at design-time. 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 4). 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 corresponds to mandatory features (Figure 3e). Otherwise, as design-time variability, other OR decompositions can be mapped into sets of OR-features (Figure 3d). 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 3c). The inclusive OR decomposition corresponds to a feature refined into a set of OR features (Figure 3d). 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 3b).

Constraints in feature models represent relationships among variable features that cannot be captured by feature decompositions. These constraints include, for example, *mutual exclusion* and *mutual dependency*. To help in feature selection, feature model constraints can be generated by relating features with their corresponding goals contributing (positively or negatively) to the same softgoal. For instance, if two system-delegated goals contribute positively (respec-

tively, negatively) to the softgoal S, then both their corresponding features will most likely have to be included in (respectively, excluded from) the system provided that the softgoal is of importance for that system variant. Thus, we generate a mutual dependency constraint between the two features. The constraint's label includes the strength of the softgoal contribution and the name of the softgoal to document the source of the constraint (e.g., $+$depends$[S]$, if both goals contributed positively to S). Similarly, if two system-delegated goals have opposite contributions to a softgoal, then selecting both corresponding features in a system that tries to satisfice the softgoal will be counterproductive. This will result in a mutual exclusion constraint between the two features. Thus, the constraints help in the feature selection process by accounting for stakeholders' quality concerns.

In general, to obtain a feature model constraint between two features f_X and f_Y based on the softgoal contributions of their corresponding goals, we use the following rules featuring the corresponding goals X and Y and a softgoal S. Here, $+(X, S)$ indicates that the goal X contributes positively to the softgoal S, $-(X, S)$ indicates that the goal X contributes negatively to the softgoal S, etc.

```
+conflicts[S](X, Y) <=>
 (+(X,S) AND -(Y,S) OR -(X,S) AND +(Y,S))
++conflicts[S](X, Y) <=>
 (++(X,S) AND --(Y,S) OR --(X,S) AND ++(Y,S))
+depends[S](X,Y) <=> (+(X,S) AND +(Y,S))          (1)
-depends[S](X,Y) <=> (-(X,S) AND -(Y,S))
++depends[S](X,Y) <=> (++(X,S) AND ++(Y,S))
--depends[S](X,Y) <=> (--(X,S) AND --(Y,S))
```

The constraints are parameterized by a softgoal S to indicate that they are significant only when S is important to the stakeholders. As well, the strength of the softgoal contributions implies the strength of the constraints (as shown by $++|+|--|-$). The process can be easily extended to support constraints among feature sets.

Below we present the proposed process to generate a preliminary feature model from an annotated goal model.

FeatureGenerationProcedure
input: Goal models with annotations (1) inclusive/exclusive OR (2) system/non-system goals.
output: An initial feature model with traceability established between goals and features.
procedure
For every root goal
1. if it is a non-system goal, return NOP
2. if it is AND-decomposed (Figure 3a)
   map its subgoals into a set of Mandatory features
3. if it is OR-decomposed
3.1 if there is no softgoal to guide the selection of its subgoals (Figure 3e), map its subgoals into a set of Mandatory features goto step 4.
3.2 if it has both system and non-system subgoals (Figure 3b)
   map its system subgoal(s) to Optional feature(s)
3.3 if it is annotated as an exclusive OR (Figure 3c)
   map it to an Alternative feature
3.4 if it is annotated as an inclusive OR (Figure 3d)
   map it to an OR feature
4. create the mapping recursively for each subgoal
5. for each softgoal with contributions from multiple goals
   create appropriately parameterized constraints using rules (1)
end

The generated feature models reflect the fact that decompositions in goal models are much more restrictive than in feature models. Thus, we produce feature models where features have sub-features of a single type and cannot have more than one set of Alternative or OR-features. One can further group them into mixed-type feature decompositions if appropriate.

The above procedure generates a preliminary design view. In a more complex design, the system may need to facilitate the actors in its environment 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 4[1]).

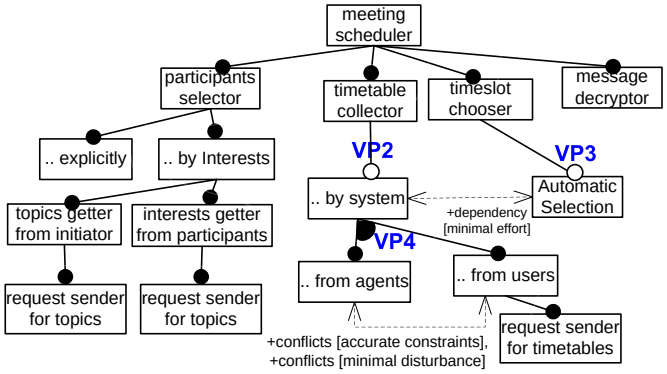

Figure 4. A feature model derived from the goal model in Figure 1

[1]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").

4 Generating statecharts

Statecharts, as proposed by David Harel [10], 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 swimlanes) or a set of *XOR* states [10]. 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.

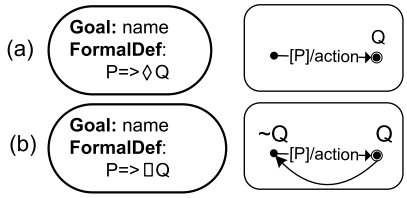

Figure 5. Mapping a leaf goal into a statechart

Figure 5 shows a mapping from a goal in a requirements goal model to a state in a statechart. In Figure 5a, an *achieve* goal is expressed as a temporal formula with P being its precondition, and Q being its postcondition. In the corresponding statechart, one entry state and one 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 by replacing the condition at the exit state with $\neg Q$. Figure 5b shows the mapping from a maintain goal to a statechart, where the mapped 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.

A goal hierarchy can also be mapped into a state hierarchy in a statechart. That is, the state corresponding to a goal becomes a super-state of the states associated with its subgoals. Here, the runtime variability in the goal model will be preserved in the statecharts through alternative transition paths.

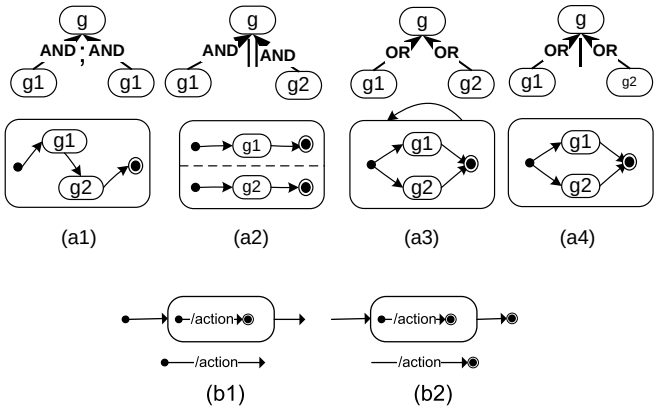

Figure 6. Statechart composition patterns

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 labelled with the action corresponding to the leaf goal (Figure 5). 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 annotations, e.g., for an OR decomposition, one has to consider whether it is inclusive or exclusive (see Figure 6).

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 5. 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 in Figure 6 can then be used to combine the statecharts of subgoals into one statechart. Specifically:

1. When a goal is AND-decomposed sequentially (`;`) into N subgoals (Figure 6a1) we create $N + 1$ transitions that connect the N subgoal states with the entry and exit states of the goal as a sequential chain. The decomposition of a sequential AND is achieved by a set of sequential sub-goals according to the left to right order of the goal graph[2].

2. When a goal is AND-decomposed concurrently (`||`) into N subgoals (Figure 6a2), we create N pairs of transitions that connect each subgoal state with the entry and exit states of the goal respectively. The states

[2]Note that in Statecharts semantics, the substates are XOR-decomposed since only one state can be active in the system at any given time.

are the AND decomposition of the superstate in the statechart.

3. When a goal is OR-decomposed into N subgoals inclusively (Figure 6a3), we create N pairs of transitions that connect each subgoal state with the entry and exit states of the goal respectively, and also create a cyclic transition for fulfilling any number of subgoals in the OR decomposition. The states here are the XOR decomposition of the superstate in the statechart.

4. When a goal is OR-decomposed into N subgoals exclusively (Figure 6a4), we just create N pairs of transitions that connect each subgoal state with the entry and exit states of the goal respectively. The states are the XOR decomposition of the superstate in the statechart.

As a result, given the number of subgoals to be N, the number of transitions introduced will be $N+1, 2N, N+1$ and $2N$ for the "sequential AND", "parallel AND", "inclusive OR" and "exclusive OR" patterns respectively.

The statechart generated using these patterns can be simplified when the goals are at the leaf level (Figure 6b): since no new intermediate state is introduced between the entry and the exit state, the action on the single transition will be moved to an incoming transition from the sibling entry superstate (Figure 6b1) or to an outgoing transition to the sibling exit superstate (Figure 6b2).

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 6a. The statechart is further simplified based on the patterns in Figure 6b. As a result, we obtain a statechart with hierarchical state decompositions (see Figure 7). It describes an initial behavioral view of the system.

An initial behavioral model for the preliminary design is generated as statecharts by the following procedure.

StatechartsGenerationProcedure
input: Goal models with annotations (1) inclusive/exclusive OR (2) sequential/parallel AND.
output: An initial statechart with traceability established between goals and state transitions.
procedure
For every root goal apply the patterns in (Figure 6a):
1. if it is AND decomposed with annotation (;)
map it into a statechart connecting XOR substates sequentially

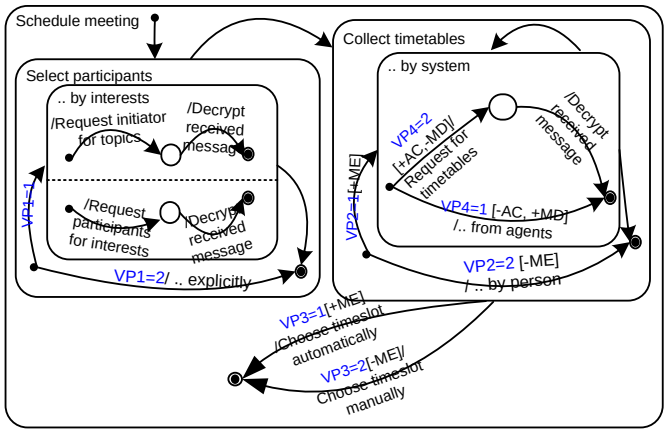

Figure 7. A statechart generated from the generic goal model in Figure 1. The softgoal names are abbreviated.

2. if it is AND decomposed with annotation (||)
map it into a statechart partitioned into parallel AND substates
3. if it is OR decomposed
map it into a statechart with each path of transition corresponding to one alternative
4. if it is annotated as an inclusive OR
add one cyclic transition on the statechart
5. recursively apply step 1-5 on every subgoal
6. simplify the leaf statecharts according to the rules in Figure 6b
end

The generated statechart can be further modified by the designer. For examples, 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 variation point selection and the softgoals used to make decisions are recorded by the guard conditions on the transitions, the changes of the statecharts can still be traced back to the composition of the annotated goal models.

5 Generating component-connector view

An important software engineering principle is to modularize a system into a set of subsystems (i.e. modules, components) that have low coupling and high cohesion [18]. The typical way to formally describe a component-connector architecture is via an Architecture Description Language (ADL). Numerous ADLs have been proposed [16]. Here, we use an adapted version of Koala [24], a simple ADL based on Darwin [15].

A representation in Koala is organized around interface types and components. An *interface type* defines a collection of message signatures as member functions with which

an implementing component can interact with its environment. A *component*, on the other hand, is defined in terms of instances of interface types (i.e. interfaces).

A PROVIDES interface shows how the environment can access the functionality that is implemented by the component, whereas a REQUIRES interface shows how the component will access the functionality provided by the environment. Usually, each REQUIRES interface of a component in the system is *bound* to exactly one PROVIDES interface of some other component. Koala allows alternative bindings of interfaces through the use of a switch. A *switch* associates one REQUIRES interface of one component with two or more alternative PROVIDES interfaces of other components, assuming they are all of the same type. Thus, a switch represents alternative bindings among interfaces. Furthermore, Koala allows the *containment* of components into compound ones.

In order to automatically produce a component-connector architecture, we will need an AND/OR goal graph and a specification of the inputs and the outputs of each goal, where applicable. Inputs are the data entities that need to be supplied to the agent responsible for the goal in order to fulfill it. Outputs are data entities that the agent provides to its environment as part of the delivery of the goal. For example, the entities "Initiator Address" and "Topics" are the input and output of the goal "Get Topics from Initiator" respectively.

The architecture is produced from a goal model as follows. We first specify the inputs and outputs of each goal. Then an *associated interface type* and an *associated component* are created for each goal. The associated interface type of a goal initially contains one operation signature, the name of which is directly derived from the description of the goal. The inputs and outputs of the goal become the IN and OUT parameters of the signature. For example, a goal "Collect Timetables" that inputs "Users" and an "Interval" and outputs "Constraints" produces the following associated interface type in Koala:

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

By default, the associated component of the goal implements the associated interface type as a PROVIDES interface. The REQUIRES interfaces of the associated component, though, are defined depending on how the goal is decomposed. If the goal is AND-decomposed, the associated component has as many REQUIRES interfaces as the subgoals. Thus in our example, the initial component of the goal "Collect timetables from users" is generated as follows:

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

In the generated component configuration, the REQUIRES interfaces are bound to the appropriate default PROVIDES interfaces of the subgoals. The name of the associated component is defined to reflect the name of the performer of the default PROVIDES interface operation.

If the goal is OR-decomposed, the associated component itself becomes a compound one. Further, the associated interface types of the subgoals are *replaced* with the associated interface type of the parent goal. Thus, the default PROVIDES interface of the parent goal is now of the same type as the default PROVIDES interfaces of the subgoals. In the generated compound component, a switch is introduced in order to bind these interfaces. Hence, the default PROVIDES interface of the component associated with the parent goal can be bound to any of the subgoal's default PROVIDES interfaces. Both the switch and the components of the subgoals are placed inside the component of the parent goal, and *hidden* behind its interface.

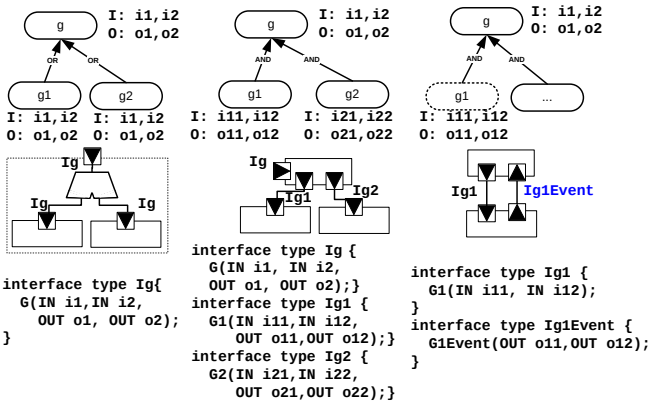

Figure 8. Patterns to generate component configurations.

The process is summarized into the following procedure.

ComponentConnectorGenerationProcedure
input: Goal model with annotations (1) inputs/outputs entities of a goal (2) whether the goal is delegated to a non-system actor
output: A configuration of the mapped components are connected through generated interfaces and switches.
procedure
For every root goal g
1. Generate a component C_g and an interface type I_g
2. Add an interface $i \in I_g$ into C_g.PROVIDES
3. if the goal g is OR decomposed (Figure 8a)
   turn C_g into a compound component;
   introduce a switch subcomponent C_g.switch;
4. if the g has a parent goal p

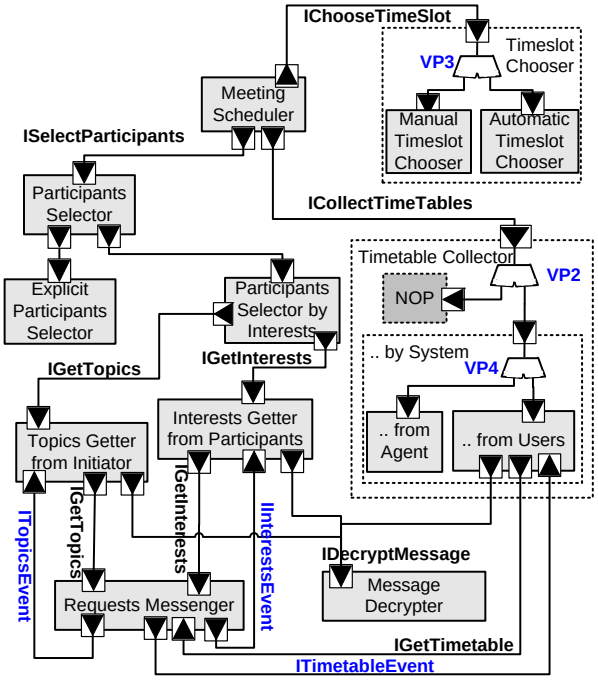

Figure 9. The generated component configuration.

4.1 if p is OR decomposed (Figure 8a)
add C_g into C_p;
create an interface of type I_p into C_g.PROVIDES;
bind the C_p.PROVIDES interface with C_p.switch.IN;
bind the C_p.switch.OUT with C_g.PROVIDES;
4.2 if p is AND decomposed (Figure 8b)
create an interface $j \in I_g$ into C_p.REQUIRES;
bind j with i
4.3 if g is a leaf goal delegated to a non-system actor
Bind an event interface in C_g.REQUIRES to
an interface in C_p.PROVIDES;
5. apply steps 1-5 recursively on the subgoals of g
end

In Figure 8, a 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 decomposition of system goals are mapped into the component-connector architecture.

In order to accommodate a scheme for *event* propagation amongst components, we follow an approach inspired by the C2 architectural style [16], where requests and notifications are propagated towards the opposite directions. In our case, as requests flow from high level components to low-level ones, notifications (*events*) originated from low level components propagate to higher-level ones. Specifically, leaf level goals can be delegated to non-system actors. In this case, the derived components are responsible for supporting the external actors' activity to attain the associated goal, sense the progress of this attainment and communicate it to the components of higher level by generating the appropriate events. We name such components *user interface (UI) components* to signify that they lay between the core of the system and the activities of the environment. UI components will introduce an additional REQUIRES interface that channels the events that the component produces. This interface will be bound to an additional PROVIDES interface at the parent component, which is set as the default handler. A typical binding example is a Listener interface in Java, implemented in the parent component for receiving events from the UI Component. The naming convention `I[GoalName]Event` can be used to distinguish event interfaces.

In our example, three goals "Send request for topics/interests/timetable" are delegated to external actors (e.g. initiator, participants and users), and will therefore yield a UI 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 default configuration has been produced.

6 Tool support and A case study

We tested the applicability of our framework in two stages. In the first stage we developed the goal model of an exemplar requirements specification problem, the meeting scheduler. We developed a model of 73 goals, 13 softgoals, 68 AND/OR decomposition links and 36 correlation dependencies. We then applied the transformation procedures in order to produce the design views. The technical soundness that the result demonstrated, served as an initial "sanity check" of our derivation procedures that were implemented using Eclipse Modeling Framework (EMF).

A basic goal model is represented by a `goal` class, which has a `subgoal` relation associating a *parent* with zero to many *children*. A goal is associated with a `DecompositionType`. If it is decomposed, the decomposition type is either `AND` or `OR`, otherwise, its decomposition type is `LEAF`. A goal can be associated with zero

to many contribution rules to a target goal. Each contribution rule is one of the `ContributionType`: `Help`, `Hurt`, `Make`, `Break`. Under the reasoning process, a goal is associated with a satisfaction label. The `LabelType` is one of `Satisfied`, `Denied`, `PartiallySatisfied`, `PartiallyDenied`, `Conflict` or `Unknown`.

We extend the goal model by adding design annotations that are sufficient to generate the three design views presented in the previous sections.

For feature models generation, we extend a `Goal` into a `FeatureAnnotatedGoal`, indicating the configuration variability, i.e., whether the goal is done by the system or not. Besides the `Feature` hierarchy, our process will produce the feature model with the constraints derived from the associated `SoftgoalRefinements`.

For statecharts generation, we extend a `Goal` into a `StateAnnotatedGoal`, indicating the pre-/post-conditions of the goal. In addition, the `GoalRefinement` is associated with a `StateRefinement`, indicating the control variability, i.e., whether the AND composition is sequential or parallel. The `FeatureAnnotatedGoal` can also be used to tell whether the OR composition is inclusive or exclusive. Our process will further generate Transitions based on these annotations.

For the component models generation, we extend a `Goal` into a `ComponentAnnotatedGoal` which provides *input* and *output* information for the goal. Such information can be used to generate for the goal an `Interface` to bind with its parent goal through the `requires` and `provides` associations. Our process will generate the specification for each Component in the Koala ADL.

By the extensibility of the model through the inheritance of a `Goal` and its `Refinement`, our representation is not limited to the discussed design views. Without making radical changes to our EMF model, one can implement other views, such as an aspect-oriented view [26] or a service-oriented view [28], by adding appropriate semantic information and mapping processes.

In EMF, a simplified goal model can be generated from the code listed in the appendix 7. The enriched goal model can be generated from the extended model by using the properties.

At a second stage we compared the derived design views with a real design of an existing email system. We analyzed the user goal *Prepare an Electronic Message*, and developed a goal model of 48 goals with 10 AND and 11 OR decompositions. Then we derived the design views accordingly. At the same time, we considered Columba [7, 27], an open source e-mail client written in Java, and identified the subset of its source code that implements its message preparation goal.

The derived feature view consists of 39 features where 9 non-system goals were turned into NOP. Among the 11 variation points, 2 of them were turned into mandatory feature decompositions because there is no apparent softgoals associated with them as the selection criteria.

The derived statecharts view consists of 21 "leaf-level" states, which would provide an accurate representation of the admissible message composition behaviors of Columba, without the need of significant changes.

The component-connector view consists of 33 concrete components controlled by 9 switches. However, as expected, they did not directly represent Columba's architecture. Thus, our investigation focused on whether Columba can potentially be re-engineered so that its design fits to the component-connector view. The evidence that we found indeed support this claim. For every component that appeared in the generated component-connector view, we were able to identify one or more Java classes that could be seen as the detailed implementation of that component. For example, the `org.columba.core.addressbook.gui.SelectAddressDialog` class is identified as part of the `AddressBook-based Capture` component. It was also observed that the components derived from goals delegated to user (i.e. The UI Components) can be associated with Java classes implementing visual controls such as `JButtons` or `JTextComponents`.

More interestingly, for most of the derived interfaces we were able to identify functions that could have implemented them. These functions were members of classes that were considered as implementations of the associated components. For example, the function `org.columba.addressbook.gui.autocomplete.AddressCollector.getMatchingOptions(String)` which searches the address book for auto-completion candidates can be identified as the default member of the interface `IFindandShowSuggestions` which was derived from the goal model through our procedures. Further, interfaces associated with Java visual components follow the same pattern as the one that "UI Components" of the component-connector view do: interfaces that are "incoming" to the UI Component are used for accessing data (e.g. `javax.swing.text.JTextComponent.getText`) whereas "outgoing" interfaces identify with Java "listener" interfaces (e.g. `CaretListener`) which the higher level components implement in order to sense the events that originate from the corresponding component. We found that in most cases one of the two interfaces associated with a "UI Components" was not needed in the design. For example, no data acquisition was needed for buttons, whereas events produced by textboxes were rarely of interest by the components of the higher level, in the part of Columba that we studied.

Our study of Columba also showed how our goal ori-

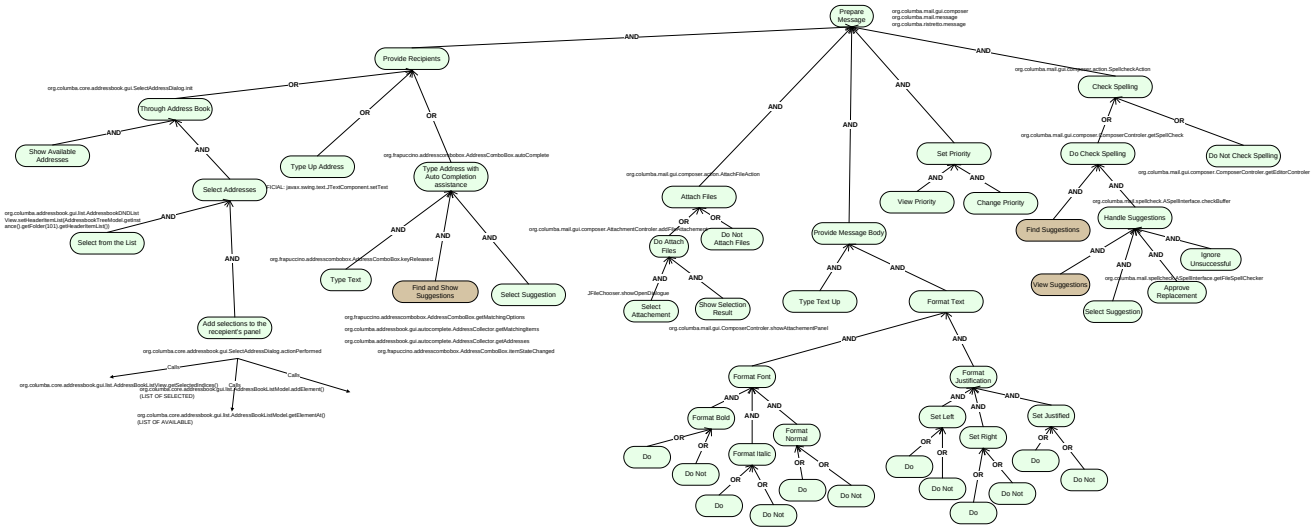

Figure 10. A generic goal model for preparing messages. Annotated are the corresponding components in Columba.

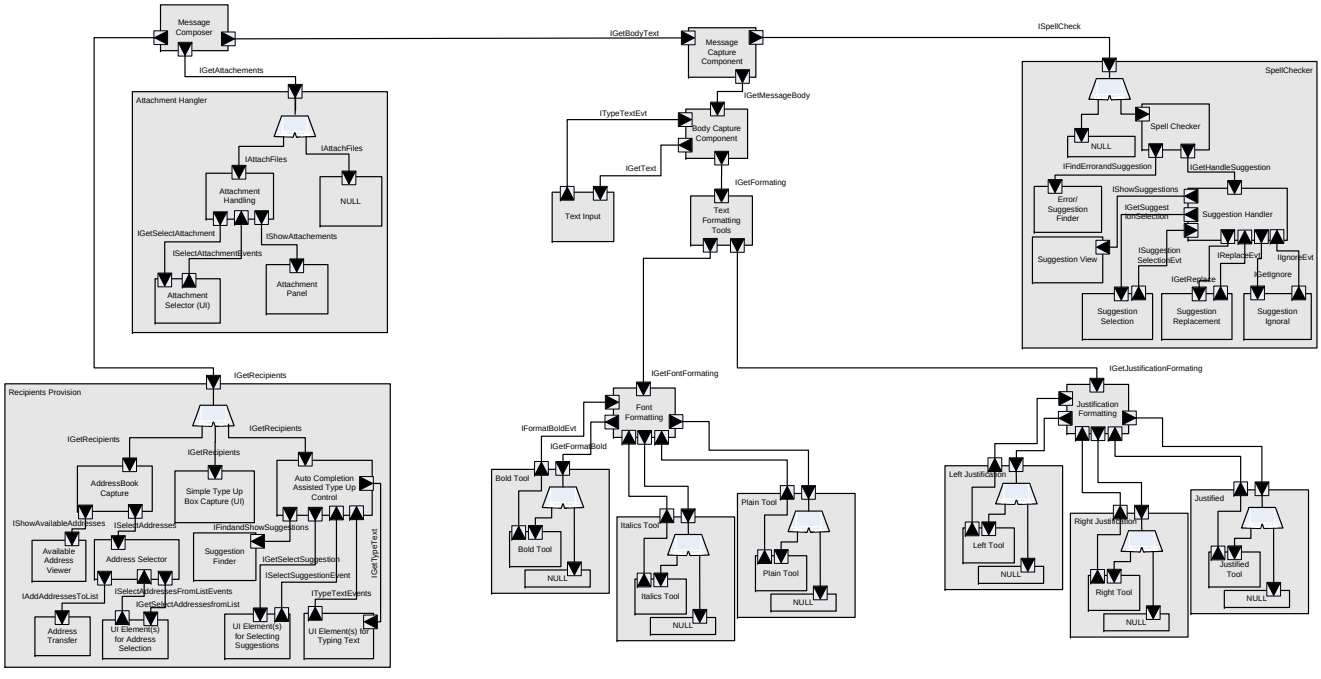

Figure 12. A component-connector configuration for preparing messages

ented approach leads to a design whose customizability is well grounded on the requirements. In our goal models, for example, having the e-mail address being auto-completed while being typed is a function some users may not prefer, because it hurts performance and introduces a mistake probability. In Columba, however, where apparently such analysis did not precede the design, auto-completion is a function that cannot be trivially taken away. The specially designed `ComboBox` which implements the auto-completion feature needs to be replaced by a plain `TextBox`. But its interaction with the rest of the system needs to be carefully studied before this replacement is possible. In our design, on the other hand, there is simply an interface for collecting addresses from the address provision component; whether auto-completion is included or not in the implementation of the interface is hidden behind the switch which directs the binding to alternative components. Thus, the newly introduced `TextBox` only needs to implement the generic interface and take its place behind the switch.

Although reengineering Columba to cleanly comply with the derived component-connector view would require significant effort, our observations clearly suggest that this would be possible, and that it would lead to a more customizable e-mail system.

7 Related work and conclusions

There is growing interest on the topic of mapping software requirements to architectures. Brandozzi et al [1] first tried to link goal oriented requirements with software architectures. They recognized that requirements and design are respectively in problem and solution domains. Therefore, a mapping between a goal and a component was proposed for increasing reusability. A more recent work by van Lamsweerde et al [21] 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. Unlike our proposal, this work assumes that one starts with a formal specification of the goal model, which may not always be viable. Complementary to their 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 [21] 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 issues at the design level.

Variability within a product family is another topic that is receiving considerable attention [4, 5]. Variability is captured there through generative programming or domain-specific languages. This line of research generally has not addressed the problem of linking product family variability to stakeholder goals (and the alternatives ways these can be achieved). Closer to our work, [9] propose an extension of use case notation to allow for variability in the use of the system-to-be. More recently [2], the same group tackled the problem of capturing and characterizing variability across product families that share common requirements.

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. To illustrate our proposal, we have conducted a case study using public domain software. The case study suggests that the designs generated are comparable in size (i.e., in the number of features, states or components) to the initial goal model.

We are currently integrating rule-specific tools into a development environment that supports the process. We are also planning further experiments to validate and refine the proposed process. Specifically, we would like to test the process for scalability. Preliminary work on this topic was published in [11]. In addition, we are interested in understanding better the sources of variability at the requirements level. Clearly, our proposed process deals with some of these sources, such as functional variability (i.e., there is more than one way to fulfill a given goal). None of our examples show variability in use (in the spirit of [9]) and perhaps other forms of requirements-level variability that cannot be captured by goal models. We would also like to integrate delegation variability (a leaf goal can be delegated to different actors, leading to different designs) into our process.

Appendix. The algorithms to generate design views from a goal model

Algorithm 1 *Generating Feature Models*

```
CreateFeatureModel(Goal g, FeatureType type, Feature parent) {
  if (g == NOP or g has no subgoals) return;
  gFeature = CreateFeature (g,type,parent);
  if g == AND (g1,...,gn) {
    for each gi { CreateFeatureModel(gi,Mandatory,gFeature); }
  } else /* g== OR (g1,...,gn) */ {
  if there exists gi == NOP {
    for each gi {
      if (gi != NOP) { CreateFeatureModel(gi,Optional,gFeature); }
  } } else /* all gi != NOP */ {
  if g == OR(g1|...|gn) {
    for each gi {CreateFeatureModel(gi,Alternative,gFeature); }
  } else {
```

```
    for each g_i { CreateFeatureModel(g_i, Or,g); }
  }
} } /* end of CreateFeatureModel */
```

Algorithm 2 *Generating Statecharts*

```
State createStateChart(Goal g) {
  s = CreateState(g); if (s==null) return null;
  if g has no subgoal { return s; }
  for each goal g that has n sub-goals g_1, ..., g_n {
  if g == AND(g_1; ...; g_n ) /* sequential AND */ {
    for i=1, n {
      s_i = CreateStateChart(g_i); addSubstate(s, XOR, s_i);
      if i =1 { t_0 = CreateTransition (s.entry, s_1); }
      else { ti = CreateTransition (s_i-1, s_i);
      if i = n {ti = CreateTransition (s_i, s.exit); }
    } } } else if g == AND ( g_1 || ... || g_n) /* parallel AND */ {
    for i=1, n {
      s_i = CreateStateChart (g_i); addSubstate(s, AND, s_i);
      t2i-1 = CreateTransition (s.entry, s_i);
      t2i = CreateTransition (s_i, s.exit);
    } else if g == OR (g_1, ..., g_n ) /* inclusive OR */ {
      for i=1, n {
        s_i = CreateStateChart (g_i); addSubstate(s, XOR, s_i);
        t_2i-1 = CreateTransition (s.entry, s_1);
        t_2i = CreateTransition (s_i, s.exit);
        for j=1, n {
          if (i!=j)
            t_n(i+1)+j = CreateTransition (s_i, s_j);
        }
      }
    } } else if g == OR(g_1 | ... | g_n) /* exclusive OR */ {
    for i=1, n {
      s_i = CreateStateChart (g_i); addSubstate(s, XOR, s_i);
      t2i-1 = CreateTransition(s.entry, s_i);
      t2i = CreateTransition(s_i, s.exit);
    } } else g = Enrich(g, g_1, ..., g_n); /* based on data dep. */
  return s;
}

void SimplifyStatechart(Goal g) {
  if (g has no subgoals, state s = g.getState() and there exist unique
  transitions t_0==(s.entry, s.exit), t_1==(s_1, s), t_2==(s, s_2)
  where s_1, s_2 are sibling states of s in the same statechart) {
    removeState(s); removeTransitions(t_0, t_1, t_2);
    Transition t = createTransition(s_1, s_2);
    t.setFunction(t_0.getFunction());
}
```

Algorithm 3 *Generating Component views*

```
Component CreateComponentView(Goal g) {
  if (g is not a system goal) return;
  i=CreateInterfaceType(g.name,g.input,g.output)
  if (g has no subgoals) {
    c = CreateComponent(); setProvides(c, i);
  } else if (g == AND(g_1, ..., g_n) {
    c = CreateCompoundComponent(); setProvides(c, i);
    c_0 = CreateComponent(); setProvides(c_0, i);
    addSubcomponent(c, c_0);
    for each subgoal g_i {
      c_i = CreateComponentView(g_i);
      addSubcomponent(c, c_i);
      p_i = getProvides(c_i); r = addRequires(c_0, p_i);
      bindInterface(r, p_i);
    }
  } else /* g == OR(g, ..., g_n) */ {
    c = CreateCompoundComponent(); setProvides(c, i);
    c_0 = CreateSwitchComponent(); setProvides(c_0, i);
    addSubcomponent(c, c_0); setRequires(c_0, i);
    for each subgoal g_i {
      c_i = CreateComponentView(g_i);
      addSubcomponent(c, c_i);
      p_i = getProvides(c_i); bindInterface(i, p_i);
    }
  }
}
```

Appendix. The annotated Java input for constructing the EMF models for the model-driven code generation

For the semantics of annotations, please refer to the documentation of the Eclipse Modeling Framework [25].

7.1 The enriched goal model

```
// goal.java
package edu.toronto.cs.goalmodel;
import java.util.List;
/** @model */
public interface goal {
    /** @model */
    String getName();
    /** @model */
    DecompositionType getType();
    /** @model */
    goal getParent();
    /** @model type="goal"
                containment="true"
                opposite="parent" */
    List getGoal();
    /** @model */
    LabelType getLabel();
    /** @model type="contribution"
```

```
            containment="true" */
    List getRule();
    /* Simple enrichments: */
    /** @model type="topic"
            containment="true" */
    List getTopic();
    /** @model default="true" */
    Boolean getSystem();
    /** @model default="false" */
    Boolean getBoundary();
    /** @model type="topic"
            containment="true" */
    List getInput();
    /** @model type="topic"
            containment="true" */
    List getOutput();
    /** @model default="true" */
    Boolean getExclusive();
    /** @model default="true" */
    Boolean getSequential();
    /** @model default="false" */
    Boolean getParallel();
    /* More enrichments: */
    /** @model type="property"
            containment="true" */
    List getProperty();
}
// contribution.java
package edu.toronto.cs.goalmodel;
/** @model */
public interface contribution {
    /** @model type="ContributionType" */
    int getType();
    /** @model */
    goal getTarget();
}
// DecompositionType.java
package edu.toronto.cs.goalmodel;
/** @model */
public final class DecompositionType {
    /** @model name="OR" */
    public static final int OR = 0;
    /** @model name="AND" */
    public static final int AND = 1;
    /** @model name="LEAF" */
    public static final int LEAF = 2;
}
// ContributionType.java
package edu.toronto.cs.goalmodel;
/** @model */
public final class ContributionType {
    /** @model name="HELP" */
    public static final int HELP = 1;
    /** @model name="HURT" */
    public static final int HURT = -1;
    /** @model name="MAKE" */
    public static final int MAKE = 2;
    /** @model name="BREAK" */
    public static final int BREAK = -2;
}
// LabelType.java
package edu.toronto.cs.goalmodel;

/** @model */
public final class LabelType {
    /** @model name="Satisfied" */
    public static final int SATISFIED = 2;
    /** @model name="Denied" */
    public static final int DENIED = -2;
    /** @model name="PartiallySatisfied" */
    public static final int PARTIALLY_SATISFIED = 1;
    /** @model name="PartiallyDenied" */
    public static final int PARTIALLY_DENIED = -1;
    /** @model name="Unknown" */
    public static final int UNKNOWN = 0;
    /** @model name="Conflict" */
    public static final int CONFLICT = 4;
}
// Property.java
package edu.toronto.cs.goalmodel;
/** @model */
public interface property {
    /** @model */
    String getName();
    /** @model */
    String getValue();
}
```

7.2 The feature model

```
/* feature.java */
package edu.toronto.cs.featuremodel;
/** @model */
public interface feature {
    /** @model */
    String getName();
    /** @model */
    DecompositionType getType();
    /** @model */
    feature getParent();
    /** @model type="feature"
            containment="true"
            opposite="parent" */
    List getFeature();
    /** @model default="false" */
    Boolean getOptional();
    /** @model type="constraint"
            containment="true" */
    List getConstaint();
}

/* DecompositionType.java */
package edu.toronto.cs.featuremodel;
/** @model */
public final class DecompositionType {
    /** @model name="AND" */
    public static final int AND = 0;
    /** @model name="Alternative" */
    public static final int ALTERNATIVE = 1;
    /** @model name="OR" */
    public static final int OR = 2;
    /** @model name="LEAF" */
    public static final int LEAF = 3;
}

/* constraint.java */
package edu.toronto.cs.featuremodel;
/** @model */
public interface constraint {
    /** @model */
    ConstraintType getType();
    /** @model */
    feature getTarget();
```

```
}
/* ConstraintType.java */
package edu.toronto.cs.featuremodel;
/** @model */
public final class ConstraintType {
    /** @model name="DEPEND" */
    public static final int DEPEND = 1;
    /** @model name="CONFLICT" */
    public static final int CONFLICT = -1;
}
```

7.3 The statechart model

```
/* state.java */
package edu.toronto.cs.statechart;
import java.util.List;
/** @model */
public interface state {
    /** @model */
    String getName();
    /** @model */
    DecompositionType getType();
    /** @model */
    state getSuper();
    /** @model type="state"
               containment="true"
               opposite="super" */
    List getState();
    /** @model */
    state getInit();
    /** @model */
    state getExit();
    /** @model type="transition"
               containment="true" */
    List getTransition();
}
/* DecompositionType.java */
package edu.toronto.cs.statechart;
/** @model */
public final class DecompositionType {
    /** @model name="AND" */
    public static final int AND = 0;
    /** @model name="XOR" */
    public static final int XOR = 1;
    /** @model name="LEAF" */
    public static final int LEAF = 2;
}
/* transition.java */
package edu.toronto.cs.statechart;
/** @model */
public interface transition {
    /** @model */
    String getName();
    /** @model */
    state getFrom();
    /** @model */
    state getTo();
    /** @model */
    String getEvent();
    /** @model */
    String getTrigger();
}
```

7.4 The components model

```
/* component.java */
package edu.toronto.cs.components;

import java.util.List;

/** @model */
public interface component {
    /** @model */
    String getName();
    /** @model */
    component getContainer();
    /** @model type="component"
               containment="true"
               opposite="container" */
    List getComponent();
    /** @model type="Interface"
               containment="true" */
    List getProvide();
    /** @model type="Interface"
               containment="true" */
    List getRequire();
    /** @model default="true" */
    Boolean getCompound();
    /** @model default="false" */
    Boolean getSwitch();
    /** @model default="false" */
    Boolean getUIComponent();
}

// connector.java
package edu.toronto.cs.components;

/** @model */
public interface connector {
    /** @model */
    String getName();
    /** @model */
    Interface getType();
    /** @model */
    component getFrom();
    /** @model */
    component getTo();
}

/* Interface.java */
package edu.toronto.cs.components;
import java.util.List;

/** @model */
public interface Interface {
    /** @model */
    String getName();
    /** @model type="variable"
               containment="true" */
    List getInput();
    /** @model type="variable"
               containment="true" */
    List getOutput();
}

/* variable.java */
package edu.toronto.cs.components;
/** @model */
public interface Variable {
```

```
   /** @model */
    String getName();
}
```

References

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

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

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

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

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

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

[7] F. Dietz and T. Stich. The Columba project, 1.0 RC2, http://columba.sourceforge.net.

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

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

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

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

[12] 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.

[13] 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.

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

[15] 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.

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

[17] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, 18(6):483–497, Jun 1992.

[18] D. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1253–1058, 1972.

[19] R. Prieto-Diaz. Domain analysis: An introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2):47–54, 1990.

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

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

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

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

[24] R. van Ommering. Koala, a component model for con-sumer electronics product software. In *ESPRIT ARES Work-shop*, pages 76–86, 1998.

[25] www.eclipse.org. Eclipse 3.0.1, 2005.

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

[27] 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.

[28] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *IEEE Trans. Software Engineering*, 30(5):311–327, 2004.

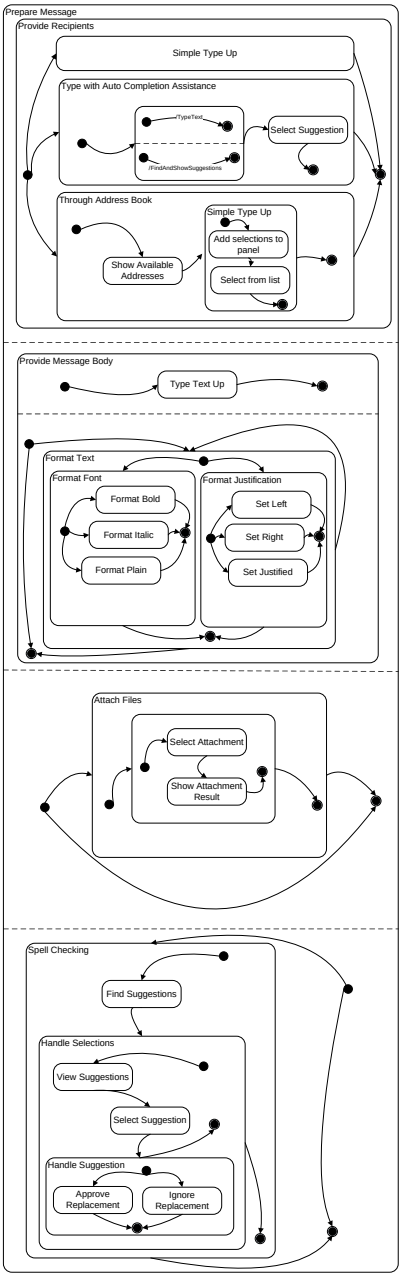

Figure 11. Statecharts for preparing messages