From Stakeholder Goals to High-Variability Software Designs

Yijun Yu[1], John Mylopoulos[1], Alexei Lapouchnian[1], Sotirios Liaskos[1],
Julio Cesar Sampaio do Prado Leite[2]
[1]Dept. of Computer Science, Univ. of Toronto, {yijun,jm,liaskos,alexei}@cs.toronto.edu
[2]Dept. of Computer Science, PUC-Rio, julio@inf.puc-rio.br

Abstract

Traditionally, software requirements consist of a list of desirable functions to be accommodated by the proposed software system. Through goal-oriented requirements engineering, stakeholder goals are analyzed into goal models that concisely define a space of alternative sets of functional requirements. We adopt this framework and propose a systematic generation of generic (high-variability) software designs that can accommodate ALL alternatives for the fulfillment of these stakeholder goals. In this paper, we enrich goal models with design-related annotations to generate three views of high-variability software design: feature models, statecharts, and component-connector models. Our process has been applied to an extensive study of the meeting scheduling problem, from which an initial high-variability design for the system-to-be is derived.

Keywords *Goal-oriented requirements engineering, goal models, software variability, software architectures, feature models, statecharts, architectural description languages*

1 Introduction

Traditionally, requirements consist of a list of functions the system-to-be should support [6], along with informally stated qualities (or non-functional requirements, NFRs [3]). In goal-oriented reequirements approaches [24, 19], early requirements represent stakeholder goals to be fulfilled by the system-to-be, and a list of qualities that serve as criteria for selecting a solution that fulfils these goals [18]. Goal models were proposed as vehicles for bridging "early requirements" with "late" ones (KAOS Project [26]). In goal models, root-level (stakeholder) goals are refined into leaf-level goals that model requirements, or tasks that system or external actors have to perform. The requirements engineer has then to choose a set of leaf-level goals which together describe a single solution to the problem.

We view the requirements as something more: our goal models characterize a space of alternatives for meeting stakeholder needs. The alternatives' space can be used as a basis for designing high variability software. There is a growing need for such software, due to the raise of applications where software has to accommodate an unpredictable set of operational environments and users (e.g., web services, peer-to-peer services, homecare software). In order to map a stakeholder goal model into high-variability design – while taking into account a set of qualities – we propose a systematic process to transform goal models into an initial software design that supports high-variability. The process generates three complementary design views: a feature model, a collection of statecharts and a component model. The feature model prescribes the system-to-be as a variable combination of configurable features. The statecharts provide a view of the alternative system behaviors. Finally, the component model describes variable structural bindings of software components.

In this work, the stakeholder goal model is treated as the logical view that underlies design views, similar to the global view in the 4+1 views [15] of the *Rational Unified Process*. To derive design views from a goal model, we propose light-weight annotations that represent design-level choices.

The rest of the paper is organized as follows: Section 2 introduces goal models with an example; Section 3 presents a variability view represented in terms of feature models, Section 4 presents a behavioral view modeled in terms of statecharts, while Section 5 presents a structural view as generated component models; Section 6 discusses tool support for the process of generating designs from a given goal model; Finally, section 7 contrasts our results with the related work and concludes.

2 Goal-oriented requirements

A *goal model* is an AND/OR graph where a goal node is refined into a number of subgoal nodes through either AND- or OR-decomposition links. Every goal has a name. A hard goal has a truth value to indicate whether it is satisfied (`true`) or denied (`false`). A soft goal has a multi-value

label to indicate the degree of its satisficing and denial: fully satisfied (FS), partially satisfied (PS), fully denied (FD) or partially denied (PD). An AND/OR decomposition of a goal G into its subgoals $G_1, \ldots, G_n$ $(n > 1)$ is denoted by either $\mathsf{AND}(G_1, \ldots, G_n) \Rightarrow G$ or $\mathsf{OR}(G_1, \ldots, G_n) \Rightarrow G$ respectively. The logic rule has 2-value logic semantics for hard goals and multi-value logic semantics for softgoals. To reason about softgoals, not only their AND/OR decomposition rules are interpreted by the multi-value logic, but new label-propagation rules such as help (+), hurt (-), make(++), break(- -) are introduced [8] as well.

This simple language is sufficient for modeling problems during early requirements, covering both functional and non-functional requirements in terms of hard goals and softgoals (quality criteria). Thus, non-functional requirements (NFRs) are treated as the first-class citizens in our framework [3].

The treatment of NFR naturally leads to system alternatives addressing various quality concerns of stakeholders, i.e. the alternatives are compared on the basis of their contributions to softgoals. Alternative solutions to a goal model arise thanks to OR decompositions. As we present in the paper, at a later stage, alternatives are accounted for in different views as configuration variability in feature models, behavioral variability in statecharts, and structural variability in components.

Throughout the paper, we use the *Meeting Scheduler* example to illustrate the proposed processes [25]. In order to "schedule a meeting" (a stakeholder goal) one needs to "collect timetables" *and* "choose schedules". Each of the subgoals has two alternative solutions, either done "manually by person" or "automatically by system". A system can collect a timetable "from agents" or directly "from users", which can be done by "sending requests" and "receiving responses".

Quality attributes such as "minimal (scheduling) effort", "good quality schedule", "minimal disturbance" and "accurate (timetable) constraints" are represented as softgoals. They can be broken down into subcriteria. For example, the "minimal effort" softgoal can be achieved by minimizing "collection effort" and minimizing "matching effort". Similarly, "good quality schedule" is guaranteed by having "minimal conflicts" and "good participation". Apparently, collecting timetable manually is a tedious task for a meeting scheduler, thus it hurts the criteria to minimize "collection effort". Such correlations can be explicitly expressed in the goal model (Figure 1).

Goals can be formally represented using a linear temporal logic (LTL) formula as in KAOS [25]. Using such a formal language, one can either define a goal, or give preconditions for its fulfillment. These preconditions may involve the availability of certain data (e.g., a participant list for a "schedule meeting" goal). Moreover, the fulfillment of

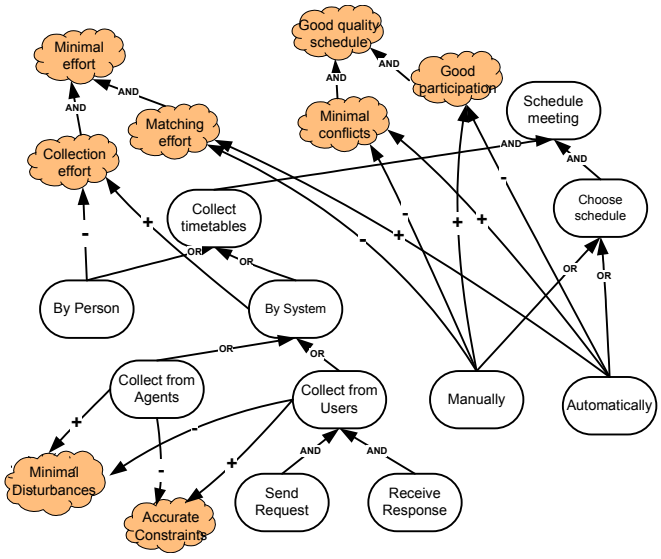

Figure 1. A goal model shows interdependencies among goals, qualities

a goal may result in the creation of information. For example, scheduling of a meeting results in the creation of time and location information for the scheduled meeting. Such conditions define input and output data for the design-level processes that operationalize a goal.

In the following three sections, we discuss the generation of three design 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 information is needed to generate that view from a 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 [21]. Thus, the software reuse community has long been interested in analyzing the commonality among closely-related software systems which consitute a *product line*. One such method is the Feature-Oriented Domain Analysis (FODA) [12]. A *feature* represents system functionality realzed by a software component. Hence, a feature constitutes a design-level concept. FODA assumes that features can be the basis for analyzing and representing commonality and variability of applications in a domain [13]. A *product line* is defined in terms of a feature model which represents variability within that family. The concept of a

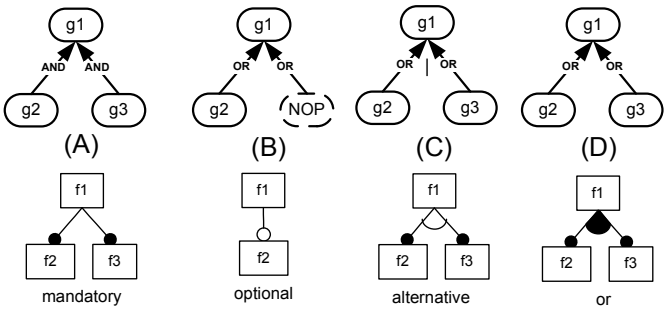

Figure 2. Goal model patterns and their corresponding feature types

feature is also popular in the software industry. For example, the Eclipse framework structures its design as a collection of plugin and fragment components that are grouped together into features [28]. Eclipse product features are organized into trees starting from the root feature that represents the entire product. The capability to group feature hierarchically allows products to be stacked using a "Russian doll" style.

There are four types of features in feature modeling (see Figure 2): Mandatory, Optional, Alternative, and Or [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.

Feature models represent configuration variability in the solution domain, whereas goal models capture variability in the problem domain, i.e., the ways that stakeholder goals can be achieved. The generation of a feature model from a goal model produces a representation that is well understood by and familiar to the software reuse community. Moreover, since feature modeling is a domain analysis technique, it is part of an encompassing process for developing software for reuse (referred to as Domain Engineering [5]), and thus can directly help in generating domain-oriented architectures and software components [13].

Goal models model how the system-to-be and its environment can together achieve root-level goals. Feature models, on the other hand, are only used to represent variability within the system-to-be. Therefore, in order to be able to generate feature models, we need to identify the subset of a goal model that is intended for the system-to-be and then map it into a feature model.

First, we need to know which leaf-level goals are assigned to the system-to-be and which are assigned to the actors in its environment. Given an initial goal model and such an assignment, where leaf-level goals to be achieved by the system's environment are replaced with NOP (no operation) goals, we can identify parts of the goal model that are not assigned to the system and must not be mapped into features. We replace a non-leaf goal with an NOP goal to indicate that it is not the responsibility of the system if all of its subgoals are NOP goals.

Next, every non-NOP goal node can be mapped into a feature with the same name. It is now easier to see that AND/OR decompositions of goals (Figure 2a/2d), if mapped into features, produce sets of Mandatory and OR-features respectively. However, Alternative and Optional feature sets do not have counterparts in the AND/OR goal models. Thus, in order to be able to generate these types of features we need to annotate goal models. First, we 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 annotated OR decomposition corresponds to a feature refined into a set of alternative features. Similarly, to produce optional features we identify patterns where a goal is OR-decomposed into a number of subgoals with at least one subgoal (NOP) being delegated to an agent in the environment of the system-to-be (Figure 2b). Then, the non-NOP sibling subgoals will be mapped into optional features. The generated feature models reflect the fact that decompositions in goal models are more restrictive than in feature models. Thus, we produce feature models where features must have subfeatures 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.

Constraints can be used in feature diagrams to represent relationships among variable features that cannot be captured by feature decompositions. These constraints include, for example, *mutual exclusion* and *mutual dependency*. Goal models allow the analysis of alternative goal decompositions with respect to their contributions to certain quality criteria. However, feature models provide no such facility and therefore the selection of features for a member of a product line family is not guided by non-functional requirements. To alleviate this, softgoal contributions present in goal models can be used to generate feature model constraints that relate features with corresponding goals contributing (positively or negatively) to the same softgoal. For instance, if two system-delegated goals contribute positively (respectively, 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 in-

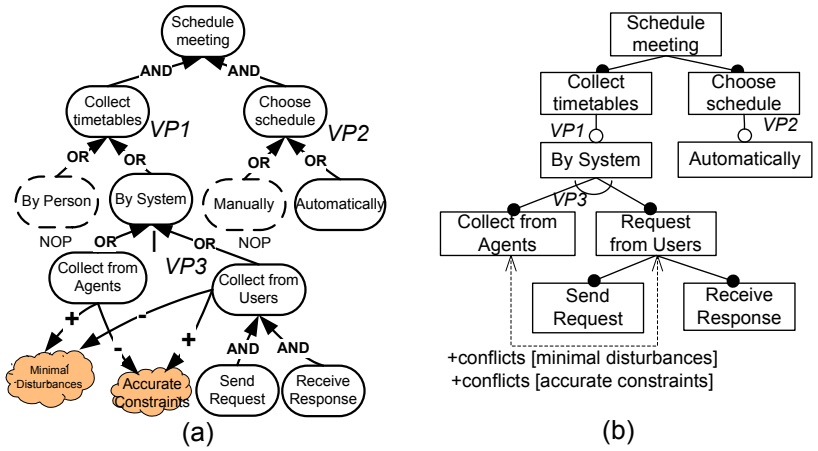

Figure 3. A feature model generated from the goal model in Figure 1

cludes the strength of 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 constraint between two features f_X and f_Y based on the softgoal S contributions from their corresponding goals X and Y, we use the following rules. 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))
-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 determines the strength of the constraints (as shown by $++|+|--|-$). The process can be easily extended to support constraints among feature sets.

For example, the stakeholder goal model in Figure 1 is simplified into a system-only goal model (Figure 3a), then four types of features are created, and two conflicting constraints are generated based on the two pairs of conflicting contributions to softgoals (Figure 3b). In the figure, one can see the correspondence between variation points (VP) in the two models.

4 Generating statecharts

Statecharts [10] constitute a visual formalism for describing the behavior of complex systems. In addition to states and transitions adopted from state machines, statecharts introduce nested super-/sub-state structures for abstraction. Specifically, a state can be decomposed into a set of *AND* substates (visually separated by swimlanes) or a set of *XOR* substates (without swimlanes) [10]. A transition can also be decomposed into transitions among the substates.

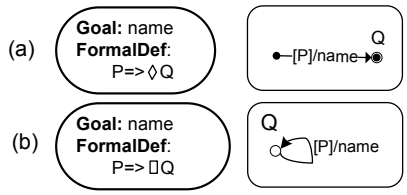

Figure 4. Mapping a goal into a statechart

Figure 4 shows a mapping from a goal in a requirements goal model to a state in a statechart. Here we explain the mapping for the four types of goals: *achieve*, *cease*, *maintain* and *avoid* on the basis of their definitions [25]. In Figure 4a, an *achieve* goal is expressed as a temporal formula ($P \Rightarrow \Diamond Q$) with P being its precondition, and Q being its postcondition. In the corresponding statechart, one entry substate and one exit substate are created: P describes the condition guarding the transition from the entry to the exit; Q prescribes the condition that must be satisfied at the exit. The transition is associated with an activity to reach

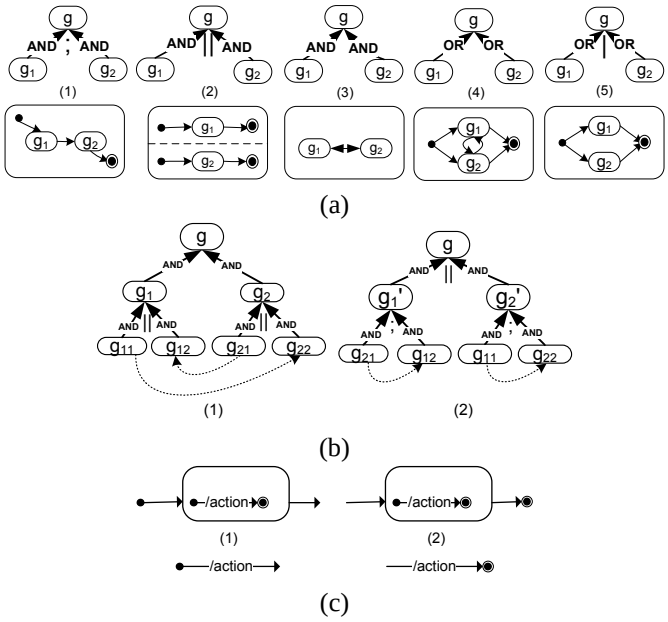

Figure 5. State decomposition patterns

the goal's desired (exit) state. The *cease* goal ($P \Rightarrow \Diamond\neg Q$) is mapped to a similar statechart by replacing the condition at the exit state with $\neg Q$. Figure 4b shows the mapping from a *maintain* goal ($P \Rightarrow \Box Q$) to a statechart, where the two substates are combined into one characterized by Q; this results in a cyclic transition, as shown in the figure. Likewise, the statechart of an *avoid* goal ($P \Rightarrow \Box\neg Q$) replaces Q with its negation.

These templates are used as a basis for the generation of an initial statechart view, without necessarily expressing the conditions as temporal logic predicates. At the detailed design stage, the designer may provide solution-specific information to specify the predicates that are required to make the refined statechart model executable.

A requirements goal model is basically an AND/OR hierarchy, where the AND decomposition of the goals are unordered and the OR decompositions are inclusive. These properties require further design-specific annotations on the decompositions in order to generate statecharts. First, the goal decomposition hierarchy is mapped into a nested statechart, i.e., the state corresponding to a goal becomes a superstate of the states associated with its subgoals; secondly, dependencies are analyzed so as to derive the order that specifies whether the subgoals can be performed in parallel or in sequence; and finally whenever possible, exclusive OR is used to simplify the combinational complexity of the inclusive OR goal decompositions.

Given a root goal, our statechart generation procedure descends along the goal refinement hierarchy recursively. For each goal, a state is created according to Figure 4. The created state has an entry and an exit substates. Next, annotations are given to the AND/OR decompositions. After that, patterns in Figure 5 are used to add transitions between the subgoal states. We use *achieve/cease* goal to illustrate the mapping patterns whereasa the *maintain/avoid* goals can be similarly composed by merging the entry and exit substates into one state. Specifically:

1. When a goal is AND-decomposed *sequentially* (;) into N subgoals (Figure 5a1) we create $N + 1$ transitions that connect the N subgoal states with the entry and exit states of the goal as a sequential chain.

2. When a goal is AND-decomposed *in parallel* (| |) into N subgoals (Figure 5a2), we create N pairs of transitions that connect each subgoal state with the entry and exit states of the goal respectively. Here the states are the AND decomposition (with swimlanes) of the superstate in the statechart.

3. When a goal is AND-decomposed into N subgoals, with neither sequential nor parallel annotation, there must be mutual dependencies among its subgoals (Figure 5b1). If the analyst knows all the dependencies among the leaf goals, then the mutual dependencies can be resolved through the restructuring that is shown in (Figure 5b2). Since such restructuring is applied bottom-up and the goal model has finite depth, in the end, all such uncertain cases can be resolved:

   Theorem 1 *Given all the dependencies among the leaf-goals, an AND/OR goal model can be restructured such that any AND-decomposed goal is either sequential or parallel composition of its subgoals.*

   Proof. Given that all the dependencies among the leaf goals are known, if one can prove the property P that after restructuring, every goal in the goal model is either a leaf-goal, an OR-decomposed goal or an AND-decomposed goal that is sequential/parallel composition of its subgoals, then the theorem is proven.
   We construct the proof bottom-up:

   (a) All the leaf goals are not decomposed, thus they satisfy P.

   (b) Suppose for all the goals at the last n depth are restructured such that the sequential/parallel decomposition is known. For any goal at depth $n + 1$ (a parent goal) that is AND-decomposed into m subgoals, if the AND-decomposition is known as sequential or parallel, then there is no need for restructuring. Otherwise, as it is not parallel, there exist dependencies among its subgoals, and as it is not sequential, there exists at least a mutual dependency among two subgoals

g_1 and g_2 such that one can not decide which one is executed first. Essentially, g_1 and g_2 must have been annotated as parallel decomposition, otherwise, if, e.g., g_1 is a sequential decomposition, then the mutual dependency among subgoals of g_1 and g_2 will lead to an impossible cyclic dependency. In this case, we consider the following restructuring. The subgoals of g_1 and g_2 are first partitioned into disconnected sets in the dependency graph: since both g_1 and g_2 have parallelism, the combined dependency graph of the subgoals of g_1 and g_2 can be partitioned into parallel connected subgraphs where each subgraph is executed sequentially. Based on the partitions, the restructuring marks g as a parallel decomposition of g_1' and g_2' as they represent the new partitions. Figure 5b1 and 5b1 show an example of such restructuring.

If the goal g is OR-decomposed, one does not need to mark it either sequential or parallel, as the subgoals are alternatives. To facilitate the restructuring of its parent goal, one can propogate the dependencies among each alternative subgoal up.

(c) Since the goal model has finite depth, the bottom-up restructuring must terminate at the root goal. Thus the property P holds for all the goals.□

However, if a goal is decomposed into leaf goals with unknown dependencies due to the lack of detailed information, a shorthand transition will be generated as an indicator of unknown mutual dependencies, so that the statechart can be completed later (Figure 5a3).

4. When a goal is OR-decomposed into N subgoals *inclusively* (Figure 5a4), 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 transition for each pair of intermediate states.

5. When a goal is OR-decomposed into N subgoals *exclusively* (|) (Figure 5a5), we just create N pairs of transitions that connect each subgoal state with the entry and exit states of the goal respectively.

As a result, assuming the number of subgoals is N, the number of transitions introduced will be $N+1, 2N, N+N^2$ 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 5c): 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 5c1) or to an outgoing transition to the sibling exit superstate (Figure 5c2).

For the "schedule a 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 "have updated time table" to "collect the updates" because the time table needs to be updated before it is collected. Similarly, "collect timetables" needs to be done before "choose schedule" because the timetables need to be collected before they are used in choosing a schedule. Secondly, we identify the inclusive/exclusive patterns for the OR-decompositions. For example, the "collect timetable by person" goal is OR-decomposed into "by email" and "by all means" inclusively, whereas the "choose schedule" is done either "manually" or "automatically". Then we add transitions according to the patterns in Figure 5a. The statechart is further simplified according to the patterns in Figure 5b. As a result, we obtain a statechart with hierarchical state decompositions (see Figure 6b). It describes an initial behavioral view of the system. The preliminary statechart can be further refined using information specific to the design stage. For example, the state "have updated timetable" can be further decomposed into a set of substates such as "have updated time from participant" for every participant; if the goal "collect timetables by all means" has been tried, s/he may not need to "collect timetables by email" again. As one can see in our more detailed case study, the refinements of the statecharts can still be traced back to the annotated goal models.

5 Generating component models

A fundamental software engineering principle is to modularize a software system into a set of subsystems (i.e. modules, components) that have low coupling and high cohesion [20]. The typical way to formally describe a component model view is via an Architecture Description Language (ADL). Numerous ADLs have been proposed [17]. Here, we use an adapted version of Koala [27], a simple ADL based on Darwin [16].

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

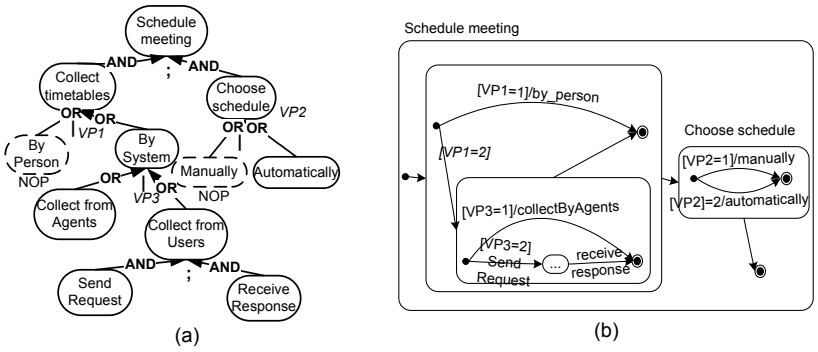

Figure 6. The statecharts generated from the goal model in Figure 1

interface of some other component. Koala allows alternative bindings of interfaces through the use of a switch. In general, 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 compatible type. Thus, a switch represents alternative bindings among interfaces. Furthermore, Koala allows the *containment* of components into compound ones. If a switch is within a compound component, its REQUIRES interface must be compatible to all the contained subcomponent's PROVIDES interfaces.

In order to automatically produce a first draft of a component model, we need an AND/OR goal decomposition tree 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 fulfilment of the goal. For example, the entities "User Name" and "User Address" are the input and output of the goal "Find Address" respectively.

The component model 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 member function 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);
}
```

As an instance of the associated interface type, the *default PROVIDES interface* of the goal is added to the associated component of the goal. 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 interface types of its subgoals. Thus in our example, the component of the goal "Collect Timetables" is generated as follows:

```
component CCollectTimetables {
    provides ICollectTimetables ct;
    requires IHaveUpdatedTimetables ht,
             ICollectUpdates cu;
}
```

In the generated component, the REQUIRES interfaces are bound to the appropriate default PROVIDES interfaces of the subgoals. Also, a compound component is created that contains both the component of the parent goal and the associated components of the subgoals.

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 one 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 7 shows the result of this process using a graphical notation directly adopted from Koala. 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

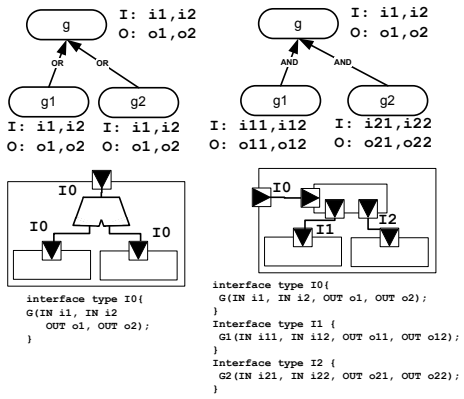

Figure 7. Patterns to generate the Koala component models

represents a switch (see Figure 13).

After the initial components model generation described above, technical intuition from the designers needs to be applied to complete the representation. First, each interface type and component need to be named after the associated goal. This can be done by converting the optative clause that describes a goal into a phrase that describes a component or an interface (e.g. from "Find Address" to "Address Finder"). More importantly, the designers may choose to merge two or more components into one, if they think their functionalities are too restricted to justify their independent existence. This can be done by introducing compound components and merging individual interface types into ones that contain the union of the member signatures of the original interface types. Conversely, the designers may introduce new interface types and components in order to describe functionality in more detail.

6 Tool support and detailed case study

We have designed the processes for generating the first draft of the feature models, statecharts and component models as parts of an initial software design. The goal models and the three kinds of design views were represented using the Eclipse Modeling Framework (EMF) [28]. EMF is an implementation of the OMG Meta-object framework (MOF) for the model-driven development. It supports serializing the models into XMI format to allow sharing them with other modeling tools such as Rational Rose and generating annotated Java programs that manipulate the model.

A UML class diagram (Figure 8) models the data structures in EMF that are used by our processes. A basic goal model is represented by a `Goal` class, which has a recursive `subgoal` relation associating a *parent* with zero to many *children*. This relation is modeled as an abstract association class `Refinement`. In the goal model, a `GoalRefinement` allows for AND/OR decompositions of the parent goal. A `Softgoal` is a subclass of a *Goal* that models a quality concern. It can be partially satisfied or denied. A refinement of a softgoal is called `SoftgoalRefinement`, representing one of the *help / hurt/ make/ break* contributions from a goal to a softgoal.

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 [29] or a service-oriented view [30], by adding appropriate semantic information and mapping processes.

Based on the problem description in [22], our meeting scheduler requirements case study has a more refined goal model, which has 73 goals, 13 softgoals, 68 AND/OR decomposition links and 36 correlation dependencies . A simplified goal model is shown in Figure 10, which has been annotated to allow for the mappings into the three design views. Figure 11, Figure 12, and Figure 13 show the generated design views. The variation points (VP) in the goal model are reflected in the design views through proper mechanisms: OR feature VP1 and optional features VP2 in the feature model (Figure 11), triggering conditions in the statechart (Figure 12) and switches of VP1 and VP2 in the component model (Figure 13). The three views are complementary to each other as they were derived from the same

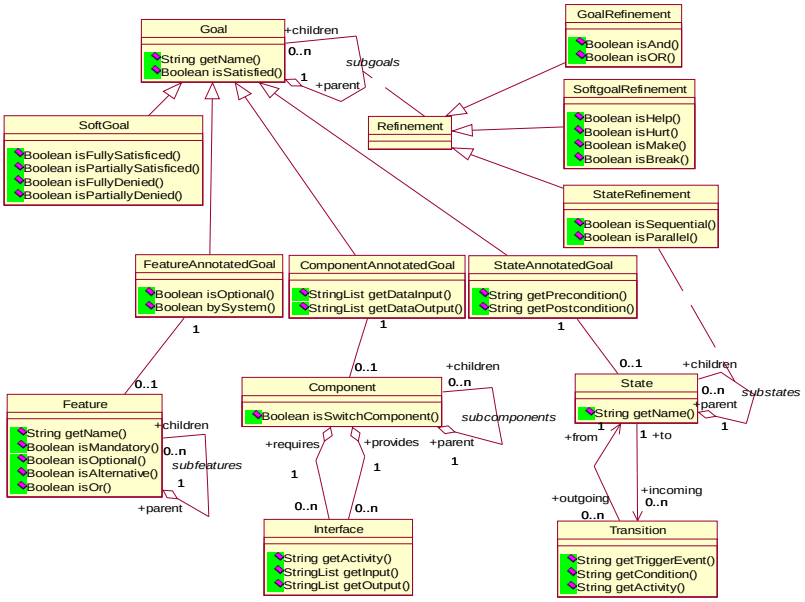

Figure 8. A data model of our system

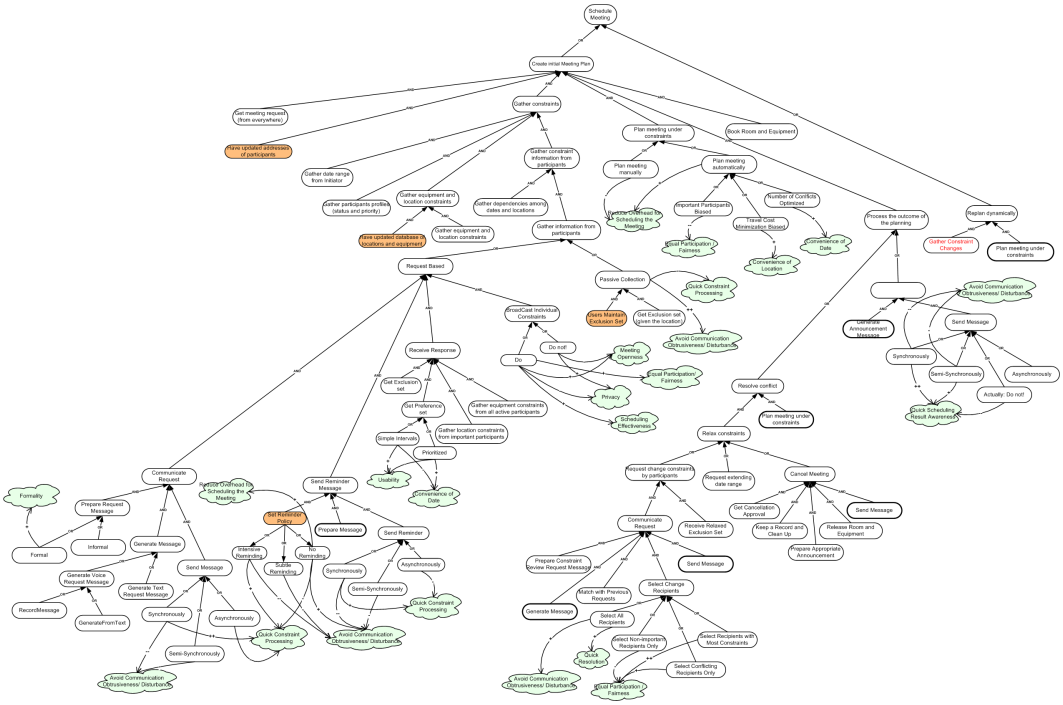

Figure 9. A detailed goal model

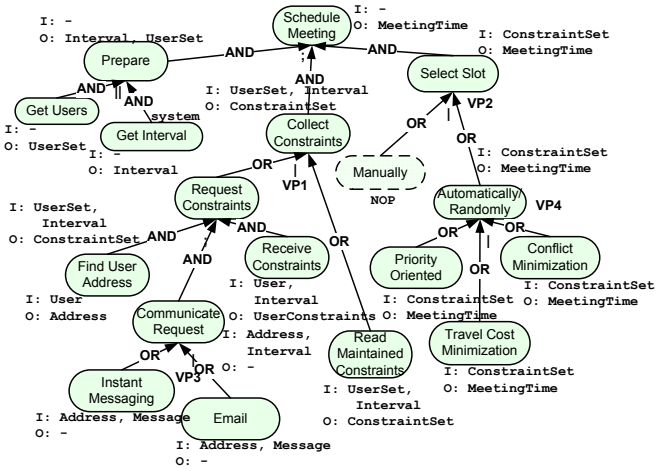

Figure 10. A refined goal model from Figure 1

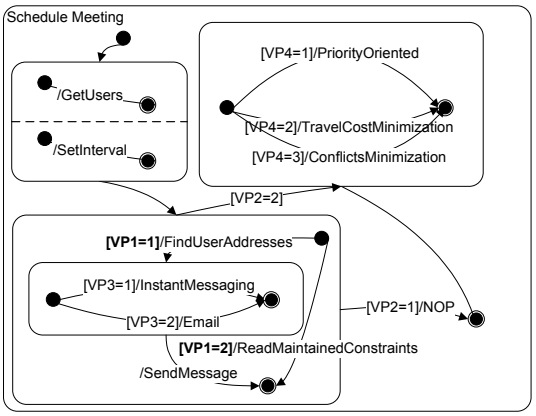

Figure 12. Statechart generated from Figure 10

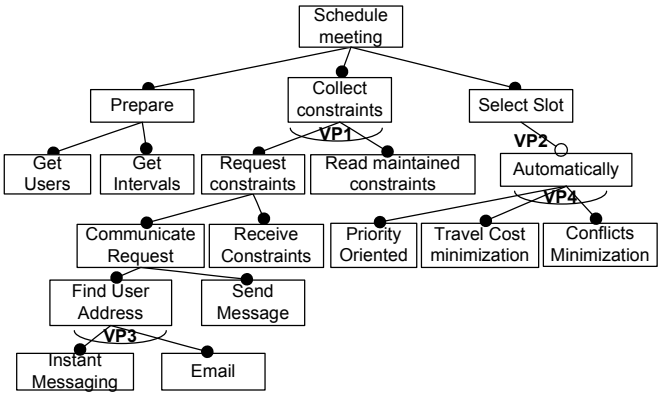

Figure 11. Generated feature model

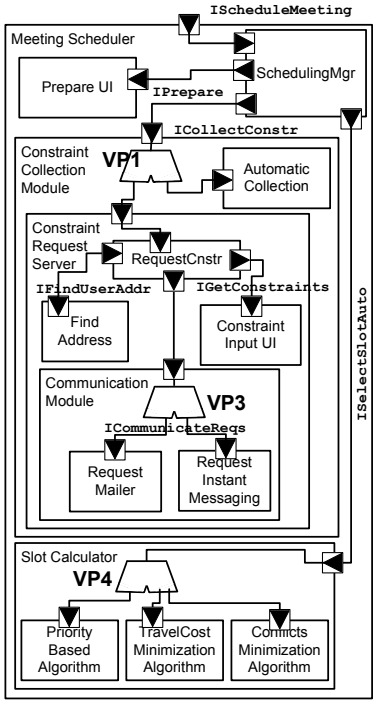

Figure 13. Component view from Figure 10

variability goal model.

7 Related work and conclusions

Models and tools for going from requirements to software architectures is a subject getting growing attention. Brandozzi et al [2] attempt 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 is proposed for increased reusability. More recent work by van Lamsweerde et al [23] derives software architectures from the formal specifications of a system goal model using heuristics, that is, by finding design elements such as classes, states and agents directly from the temporal logic formulae representing the goals. This work

assumes that one starts with a precise specification of the goals. Complementary to their work, we apply light-weight annotations to the goal model in order to derive the design views. Our work is fundamentally different from these other works in that it aims to generate a high-variability design where all solutions to a goal model are to be accomodated.

Variability has been studied either as part of requirements models or as part of design. At the requirements level [9] represents variation points in use cases whereas we use OR goals to represent them in goal models. At the design level, product-families techniques [4] can represent structural variability in module diagrams [1] or ADL [27, 7]. Configuration variability is supported in feature model through generative programming [5]. In addition, we also use statecharts [10] for behavioral variability. We show the connection between these design-level variability as they all arise from the OR goals in the early requirements. Within this paper, we do not address how to obtain a high-variability goal model. Elsewhere [11], we describe a configuration process that results in a requirements goal model with 10^{20} possible solutions for personalized email software system. The configuration process drawn from the knowledge of variable user skills and stakeholder preferences to generate the space of possible solutions. High-variability software designs generated from high-variability goal models can serve as the basis for autonomic computing [14] systems that exhibit self-configuring, self-optimizing, self-healing and self-protecting bahaviours by autonomously adapting to the changes in the stakeholder requirements and the environment.

In summary, we have proposed a systematic process for generating a high-variability design from a goal model. Our process generates three design views: feature models, statecharts and components models. The process is supported by algorithms and has been illustrated with a standard example from the literature.

For the future, we hope to refine the process and build tools that incorporate the algorithms we have proposed. We also hope to apply it to the design of real-world case studies, such as homeware that helps users with cognitive impairments live their lives (in the spirit of [11]); or web services for a telecommunications company.

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 gi { CreateFeatureModel(gi, 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 g1, ..., gn {
  if g == AND(g1; ...; gn ) /* sequential AND */ {
    for i=1, n {
      si = CreateStateChart(gi); addSubstate(s, XOR, si);
      if i =1 { t0 = CreateTransition (s.entry, s1); }
      else { ti = CreateTransition (si-1, si);
      if i = n {ti = CreateTransition (si, s.exit); }
    } } } else if g == AND ( g1 || ... || gn) /* parallel AND */ {
    for i=1, n {
      si = CreateStateChart (gi); addSubstate(s, AND, si);
      t2i-1 = CreateTransition (s.entry, si);
      t2i = CreateTransition (si, s.exit);
    } else if g == OR (g1, ..., gn ) /* inclusive OR */ {
      for i=1, n {
        si = CreateStateChart (gi); addSubstate(s, XOR, si);
        t2i-1 = CreateTransition (s.entry, s1);
        t2i = CreateTransition (si, s.exit);
        for j=1, n {
          if (i!=j)
            tn(i+1)+j = CreateTransition (si, sj);
        }
      }
    } } else if g == OR(g1 | ... | gn) /* exclusive OR */ {
    for i=1, n {
      si = CreateStateChart (gi); addSubstate(s, XOR, si);
      t2i-1 = CreateTransition(s.entry, si);
      t2i = CreateTransition(si, s.exit);
    } } else g = Enrich(g, g1, ..., gn); /* based on data dep. */
  return s;
}
```

```
void SimplifyStatechart(Goal g) {
  if (g has no subgoals, state s = g.getState() and there exist unique
  transitions t0==(s.entry, s.exit), t1==(s1, s), t2==(s, s2)
  where s1, s2 are sibling states of s in the same statechart) {
    removeState(s); removeTransitions(t0, t1, t2);
    Transition t = createTransition(s1, s2);
    t.setFunction(t0.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(g1, ..., gn) {
    c = CreateCompoundComponent(); setProvides(c, i);
    c0 = CreateComponent(); setProvides(c0, i);
    addSubcomponent(c, c0);
    for each subgoal gi {
      ci = CreateComponentView(gi);
      addSubcomponent(c, ci);
      pi = getProvides(ci); r = addRequires(c0, pi);
      bindInterface(r, pi);
    }
  } else /* g == OR(g, ..., gn) */ {
    c = CreateCompoundComponent(); setProvides(c, i);
    c0 = CreateSwitchComponent(); setProvides(c0, i);
    addSubcomponent(c, c0); setRequires(c0, i);
    for each subgoal gi {
      ci = CreateComponentView(gi);
      addSubcomponent(c, ci);
      pi = getProvides(ci); bindInterface(i, pi);
    }
  }
}
```

References

[1] F. Bachmann and L. Bass. Managing variability in software architectures. In *SSR '01*, pages 126–132. ACM Press, 2001.

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

[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] E. M. Dashofy and A. van der Hoek. Representing product family architectures in an extensible architecture description language. In *PFE '01*, pages 330–341, 2002.

[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*, 1, 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 *RE'03*, pages 117–126, 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, SEI/CMU, 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] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

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

[16] 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.

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

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

[19] J. Mylopoulos, L. Chung, and E. Yu. From object-oriented to goal-oriented requirements analysis. *CACM*, 42(1):31–37, Jan. 1999.

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

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

[22] M. Shaw, D. Garlan, R. Allen, D. Klein, J. Ockerbloom, C. Scott, and M. Schumacher. Candidate model problems in software architecture. www.cs.cmu.edu/ vit/paper_abstracts/modprb1-3.html.

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

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

[25] A. van Lamsweerde, R. Darimont, and P. Massonet. Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. In *RE'95*, pages 195–203.

[26] A. van Lamsweerde and L. Willemet. Inferring declarative requirements from operational scenarios. *TSE*, 24(12):1089–1114, Nov. 1998.

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

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

[29] Y. Yu, J. Leite, and J. Mylopoulos. From requirements goal models to goal aspects. In *RE'04*, 2004.

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