A Goal Model Elaboration for Localizing Changes in Software Evolution

Hiroyuki Nakagawa
The University of Electro-Communications
Tokyo, Japan
nakagawa@uec.ac.jp

Akihiko Ohsuga
The University of Electro-Communications
Tokyo, Japan
ohsuga@uec.ac.jp

Shinichi Honiden
National Institute of Informatics
Tokyo, Japan
honiden@nii.ac.jp

***Abstract*—Software evolution is an essential activity that adapts existing software to changes in requirements. Localizing the impact of changes is one of the most efficient strategies for successful evolution. We exploit requirements descriptions in order to extract loosely coupled components and localize changes for evolution. We define a process of elaboration for the goal model that extracts a set of control loops from the requirements descriptions as components that constitute extensible systems. We regard control loops to be independent components that prevent the impact of a change from spreading outside them. To support the elaboration, we introduce two patterns: one to extract control loops from the goal model and another to detect possible conflicts between control loops. We experimentally evaluated our approach in two types of software development and the results demonstrate that our elaboration technique helps us to analyze the impact of changes in the source code and prevent the complexity of the code from increasing.**

***Index Terms*—Software evolution, goal modeling, elaboration techniques, control loops.**

I. I

Software evolution [32] is a process where an existing software system is adjusted to satisfy a new set of requirements, and it has played a central role in the overall software lifecycle in recent years. It is generally acknowledged that the software employed in real-world environments must continuously evolve and adapt [18], [25]. Many studies, on the other hand, have suggested that software evolution consumes a large part of development costs [1], [30].

Localizing the impact of changes is a promising strategy for effectively supporting software evolution [26]. It helps us to correctly estimate the impact of changes on systems (*Impact analysis* [13], [24]) and make changes (*change implementation*) that minimize increases in code complexity [2], [15]. Both of these activities are integral to software evolution. We aim to establish a development process that can help to limit the influence of changes. Our approach is built on the idea of using requirements descriptions to bind change requirements to corresponding independent system components and thereby limit the impact of changes. This localizes the changes in the requirements description and makes it easier to understand where changes are required in the system by identifying where requirements have been changed in the descriptions. Localizing changes also prevents code from becoming too complicated as the system evolves.

This paper introduces an elaboration process that refines the goal model to help developers identify where changes are required in the subsequent development process. The elaboration process is applied to the goal models, and the process extracts *control loops* [31], [16] as software components by refactoring the goal models with our two description patterns. A control loop is a cyclic activity flow that defines independent behaviors, and we regard control loops as highly independent modules. As Nuseibeh stated in [23], software requirements affect the system architecture, and we are confident that a system with high modularity and independence will have the advantage of localizing the impact of changes. As a result, extracting control loops from the requirements description enhances the impact analysis and change implementation activities.

This paper makes two main contributions:

(1) We define an elaboration process for goal modeling that extracts control loops constituting the system. This process is intended to apply to new systems, and we localize the impact of changes in the corresponding control loops at the requirements engineering stage. We also introduce two patterns to support the elaboration: the first is to identify control loops in the goal model, and the second is to detect possible conflicts between the control loops.
(2) We report the results of applying our elaboration process to two software developments: a real-world modeling tool and a control system. This empirical evaluation revealed that in comparison with MVC and centralized control architectures, our approach improved the accuracy of the impact analysis and it limited the code complexity while reducing modification costs.

The rest of the paper is organized as follows: Section II describes how we elaborate the goal model to localize the impact of changes, and Section III explains our elaboration process for goal modeling. Section IV describes how elaborated goal models are used in software development including software evolution. Sections V and VI present the results of two experimental software developments, and Section VII explains how we evaluated our approach with the experimental results. Section VIII discusses related work, and Section IX concludes the paper.

II. A

Impact analysis and change implementation [32] are the fundamental activities in software evolution. However, it is still difficult to predict the impact of software changes even if the developers are experienced. Therefore, a software development process should help developers correctly analyze the impact of changes. Moreover, having to make continuous changes to a piece of code easily makes it complicated. Therefore, the process should also help developers to change systems while preventing code from becoming too complicated. That is, the development process should provide certain support for these activities:

- **Impact analysis:** The development process should help developers correctly estimate the impact of requirements changes.
- **Change implementation:** The development process should efficiently suppress the code complexity in order to deal with continuous changes.

To support these activities, we try to localize the changes for evolution. We use the goal model [33] as a requirements description to localize the impact of changes. This model provides goal refinement links to define the goal structure, and we exploit this structure in order to decompose the change requirements into independent blocks in the model. We also leverage the structure in order to establish traceability links [8], [14], [27], which are used to propagate localized changes in the goal model to the subsequent models.

Since high cohesion and low coupling components are beneficial for localizing the impact of changes, the components of the system should be designed separately and made as independent as possible. We devise a technique of refining the goal model to derive individual blocks from the requirements. Since the goal model represents the states to be achieved by the system, the assignment of responsibilities from the behavioral viewpoint seems to be suitable for such a derivation. We focus on *control loops* and regard them as behavioral blocks in the goal model.

A control loop is a cyclic activity flow based on the *process control model* [31], which is a model for defining system behaviors. A controller in the model forms a (control) loop consisting of four key activities: *collect, analyze, decide*, and *act*. This cycle starts with a collection of relevant data that reflects the current state of the system. The system analyzes the collected data and adjusts its behavior to achieve its goals. After that, the system acts in a way that reflects decisions. This model has three process variables to define the relationship with the environment, i.e., *input variables* that measure input, *controlled variables* that the system intends to control, and *manipulated variables* that can be changed by the controller.

We introduce an elaboration process that refines the goal model in a way that decomposes the user requirements into individual blocks corresponding to control loops. Such a refinement enables us to extract independent components that can act by themselves while providing individual functions. When additional requirements appear, new control loops are easy to embed into a system constructed with multiple control loops. Even if the requirements change, the idea of compositional control loops can easily accommodate these changes simply by updating the relevant control loops.

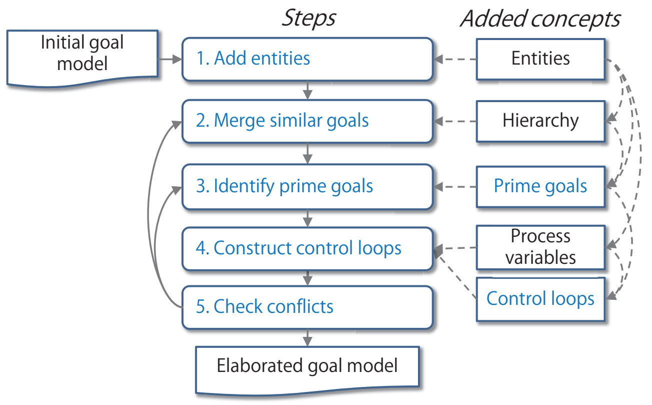


Fig. 1. Elaboration process.

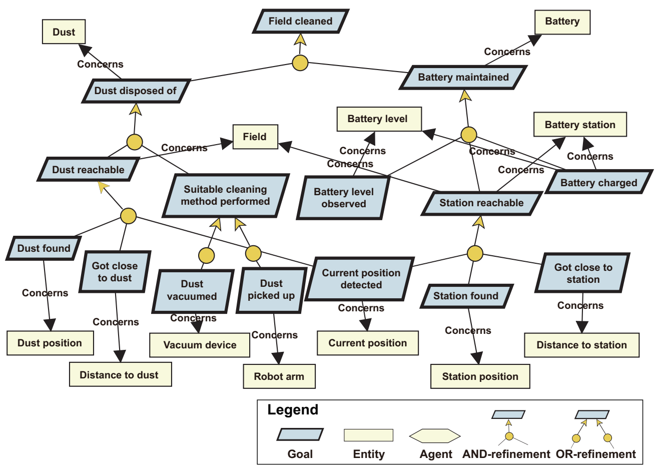


Fig. 2. Goal model after adding entities.

III. E P

Here, we define an elaboration process that helps us to determine a system's components by extracting control loops from the goal models. Figure 1 outlines the flow for the elaboration process. We use the KAOS [10], [19] method to describe the goal model in our explanation. KAOS not only has two kinds of refinement links between goals, i.e., AND-refinement links and OR-refinement links, but also several kinds of link such as those for representing *concerns* from goals to entities.

A. Preparation

We assume that the goals of the initial goal model have been adequately decomposed into subgoals with refinement links. Our process starts with two steps: *add entities* and *merge similar goals* (See Figure 1). First, we add relevant entities with *concerns* links to the goal model. Several goal modeling

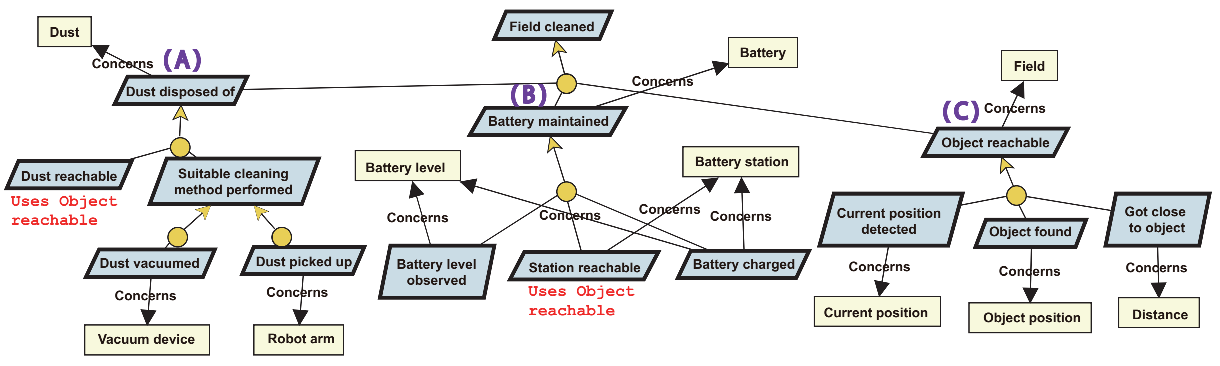


Fig. 3. Goal model of cleaning robot (after similar goals have been merged).

methods including KAOS support such an entity definition; however, we use the relationship between goals and entities for aggregating similar subgoals, identifying goals that can be assigned to control loops, and detecting possible conflicts in subsequent steps.

After entities have been added, we merge similar goals into a generalized goal. Developers should merge similar goals by considering their similarities and the similarities of their *concerns* links to entities. If two goals and all of the subgoals of each one are similar, we merge these goals and generalize them as a subtree. We use the "Uses" notation to represent dependencies on the generalized goal. If goal A needs generalized goal B[1] to achieve it, we label "Uses B" on goal A. We use this notation to explicitly separate the descriptions of generalized goals so that we can extract the control loops in the subsequent steps.

Example 1 — Here, we give a running example of the development of a cleaning robot. Figure 2 illustrates a goal model for the cleaning robot after adding entities. This goal model includes two major goals about cleaning dust items and battery maintenance. Both two goals have two similar subtrees, one is for approaching dust items (root: *Dust reachable*) and the other is for approaching the battery station (root: *Station reachable*); therefore, we merge these subtrees to a generalized subtree for approaching objects. Figure 3 illustrates a goal model after extracting the generalized subtree whose root goal is *Object reachable* (labeled (C)). Note that *Dust reachable* and *Station reachable* are given Uses labels to represent the dependencies on the new subtree.

B. Identification of Prime Goals

We extract *prime goals*, which correspond to key goals for modularity with high independence, from the goal model. We assign individual control loops to the corresponding subtrees in the goal model whose root goals are prime goals. After extracting the *prime goal candidates*, we determine which of them should be the prime goals. Our guideline for determining prime goal candidates is as follows:

[1]From here on, we call such a goal B a *generalized goal*.

Definition III.1 Guideline for Extracting Prime Goal Candidates: *Goal g_i that satisfies at least one of the following rules is defined as a prime goal candidate, where $Child_{g_x}$ is a set of child goals of goal g_x and $uses(g_x, g_y)$ is a predicate that returns true if and only if "Uses g_y" is defined on goal g_x.*

- *Rule 1. Other goals depend on g_i through* Uses *labels.*

$$\exists g_i\ (\exists g_j\ uses(g_j, g_i)) \Rightarrow\ g_i \in PGCand$$

- *Rule 2. More than one child goal (immediate subgoal) of g_i has* concerns *links to the same entities.*[2]

$$\exists g_i\ (\exists g_{ik}\ \exists g_{il}(g_{ik}, g_{il} \in Child_{g_i} \wedge g_{ik} \neq g_{il} \wedge \exists ent(ent \in Entity \wedge concerns(g_{ik}, ent) \wedge concerns(g_{il}, ent)))) \ \Rightarrow\ g_i \in PGCand \quad \square$$

Rule 1 defines generalized goals as prime goal candidates. These goals represent states that common functions, which are necessary for achieving multiple goals connected by uses dependencies, should reach. These functions are usually modularized, and thus, we regard the corresponding goals, i.e., generalized goals, as candidates. Moreover, we focus on *concerns* links from goals to entities in order to provide multiple accesses from the several blocks of the system. Rule 2 aims to localize the association with entities into single prime goals. We try to combine goals that are concerned with the same entity into one module in order to avoid unnecessary couplings of modules.

Example 2 — Figure 3 illustrates three prime goal candidates. First, the generalized goal *Object reachable* (labeled (C)) is a prime goal candidate as a result of applying Rule 1. As for Rule 2, since multiple subgoals of the goals labeled (A), (B), and (C) have concerns links to the same entities, i.e., *Dust* for goal (A)[3], *Battery*, *Battery level*, and *Battery Station* for goal (B), and *Field* for goal (C), we can extract these goals as candidates. As a result, the goals labeled (A) to (C) in Figure 3 are extracted as prime goal candidates.

[2]Here, if a child goal concerns (has a relationship with) an entity that its parent goal has the concerns link to, we regard the child to have the concerns link to the entity.

[3]As previously described, the concerns link to Dust represents multiple concerns links from the subgoals of goal (A).

Our elaboration aims at extracting multiple control loops that can construct a target system. We check which combination of prime goal candidates is adequate for assigning these control loops. The process defines the following necessary conditions that help us to select a set of prime goals from among the candidates.

Definition III.2 Necessary conditions for prime goal extraction: *A goal set PGoals whose members are picked out from the prime goal candidates ($PGoals \subseteq PGCand$) should satisfy all of the following conditions, where Leaves is the set of goals that have no subgoals and G_i is the set of all goals included in the goal tree with the root node g_i. The goals in PGoals are called prime goals.*

- *Condition 1. All leaf goals are included in a subtree whose root node is a prime goal (goal in PGoals).*

$$\forall g\ (g \in Leaves \wedge \exists i\ (g_i \in PGoals \wedge g \in G_i))$$

- *Condition 2. There are no prime goals that include another prime goal as offspring.*

$$\forall i \forall j\ (g_i, g_j \in PGoals \wedge i \neq j \Rightarrow g_i \notin G_j \wedge g_j \notin G_i) \qquad \square$$

Since control loops are assigned to individual prime goals, Condition 1 guarantees that all leaf goals are achieved by one of the control loops, and Condition 2 prevents duplications of leaf goal assignments, which cause changes to spread among control loops when the leaf goal is changed in a future evolution. If leaf goals exist that are not covered by any of the candidates, that is, if no *PGoals* can satisfy Condition 1, we add these leaf goals to the candidate set *PGCand*. If Condition 2 is not satisfied, we should exclude the upper or lower goals from *PGoals*.

C. Construction of Control Loops

After the prime goals have been determined, we assign one control loop to each corresponding subtree whose root goal is a prime goal. Since control loops can run autonomously, this assignment means that we construct independent components (control loops) from others in accordance with the extracted prime goals. In order to clarify the activities of the control loops, we refine the subgoals that are required to achieve the prime goals. We use a description pattern, called a *control loop pattern*, to refine these subgoals so that they represent the activities of the control loops.

Figure 4 illustrates a *control loop pattern* that has been extended from our previous work [22] to explicitly define the process variables. This pattern expresses the activities of the control loop — collect, analyze, decide, and act — to restructure the goal model by explicitly extracting these activities. According to this pattern, we first map the *Analyze & Decide type* onto the prime goals. This means that the prime goals are responsible for *analyzing* the current situation and *making decisions* to achieve their own goals. Next, we define process variables, i.e., *input, manipulated,* and *controlled variables*. Some of the entities correspond to these variables. If some variables are missing, we add new entities to the

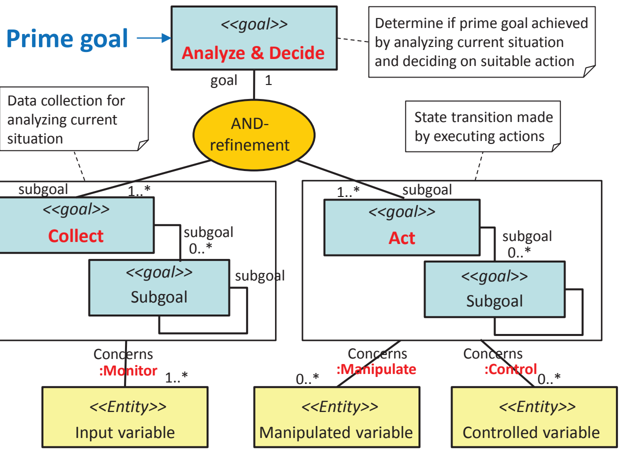


Fig. 4. Control loop pattern

goal model. These variables are explicitly defined by labeling *concerns* links, as outlined in Figure 4. After defining the process variables, we classify goals in the subtree into ones to collect information (*Collect type*) and take actions (*Act type*) by aggregating or newly defining subgoals as needed. Such additions according to the pattern may help us to find missing entities and goals.

Example 3 — Figure 5 illustrates the goal model for our cleaning robot after the control loop construction. This goal model has three prime goals labeled “(A)”, “(B)”, and “(C)”. All of these prime goals have had individual control loops embedded according to the control loop pattern.

D. Conflict Checking

Embedding multiple control loops into a system may cause *conflicts* among them. Conflicts may occur when a shared variable is changed by more than one module, and a manipulated variable is defined as a variable that a control loop can change; therefore, our elaboration particularly checks for possible conflicts between manipulated variables. [4] The elaboration process extracts such conflicts from the structure of the goal model. If conflicts are extracted, we can restructure the goal model by following our conflict resolution methods. To detect possible conflicts, we define a pattern called an *entity-conflict pattern* in the goal model.

Definition III.3 Entity-conflict pattern. *An entity ent that satisfies the following condition is a possible conflict entity.*

$$\begin{aligned}\exists i\ \exists j\ (g_i, g_j \in PGoal \wedge \exists ent\ (ent \in Entity \wedge \exists k\ \exists l\ (g_{ik} \in G_i \\ \wedge g_{jl} \in G_j \wedge\ concerns(g_{ik}, ent, ``:Manipulate") \\ \wedge concerns(g_{jl}, ent, ``:Manipulate"))))\end{aligned} \qquad \square$$

The upper goal model in Figure 6 illustrates the entity-conflict pattern that detects entities having *concerns* links labeled “:Manipulate” from multiple trees whose root nodes

[4]References to input variables do not cause conflicts. Since conflicts between controlled variables are caused by conflicts of goals, they should be resolved by using traditional conflict resolution methods from the goal model literature, such as [35].

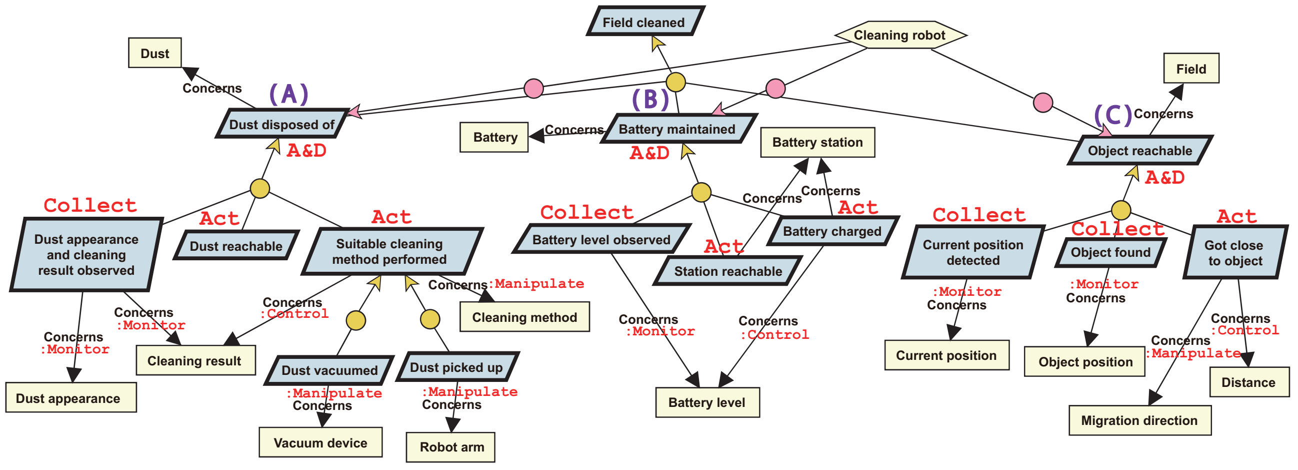


Fig. 5. Goal model after the control loop pattern has been applied. Label "A&D" represents *Analyze & Decide* type goals.

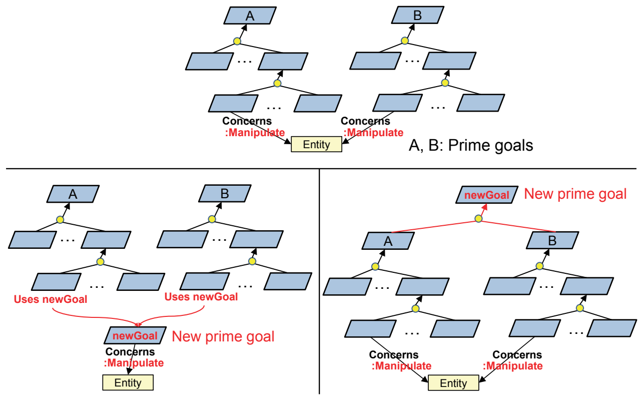


Fig. 6. Conflict detection pattern and solutions: (Upper) Entity-conflict pattern. Two lower goal models represent solutions: (Lower left) Link aggregation. (Lower right) Goal aggregation.

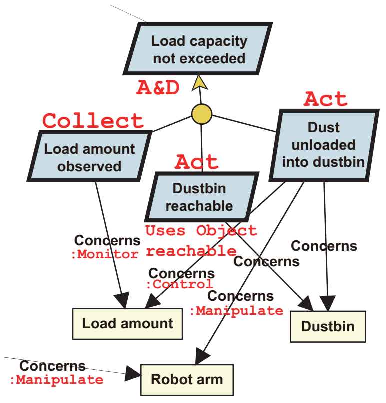


Fig. 7. Additional goals for load management. A possible conflict on *Robot arm* is detected by using the entity-conflict pattern.

are prime goals. This pattern is for extracting the manipulated variables (in Figure 4) of multiple control loops that may cause conflicts.

To resolve the conflicts detected by the entity-conflict pattern, the sources of the *concerns* links should be consolidated into the tree of a single prime goal. There are two reforming methods for resolving such entity conflicts:

- **Solution 1: Link aggregation.** *Define a new goal that has a consolidated concerns link to the entity and replace multiple concerns links with the uses dependencies of the new goal. This change means that the new goal should have the responsibility of access control of the entity.*
- **Solution 2: Goal aggregation.** *Aggregate the conflict prime goals into a tree by adding a new root goal, which becomes a new prime goal. This change means that the new goal should have the responsibility of deciding a suitable subgoal in accordance with the situation with access control.*

The lower goal models in Figure 6 illustrate these two solutions. We should choose a suitable solution from these two by considering the meaning of the prime goals. Solution 1 is appropriate for dealing with prime goals whose objectives are for providing comparatively different functions, while Solution 2 is better suited for aggregating prime goals that have similar objectives and are therefore unifiable.

Example 4 — We should consider adding a new requirement for load management into the goal model illustrated in Figure 5. This additional requirement is described as a subtree whose parent goal is *Field cleaned* in Figure 5 and forms a new control loop (Figure 7). It requires the robot's arm to be manipulated to unload dust items and causes a conflict. In this case, we can use Solution 1 in which two control loops that provide comparatively different functions use the robot's arm, i.e., one to dispose of dust items and the other to manage the load. As a result, a new control loop to manage the arm will appear.

IV. E S D

We apply the elaboration process to the initial development and to the evolution of the software. Three activities are conducted in our development.

1) *Goal model construction and elaboration*: After constructing the goal model we elaborate it by performing the process described in Section III. If requirements are changed or added, we update the goal model to accommodate them and perform the elaboration process on it again. As a result, the corresponding control loops acquire new or modified goals.

2) *Control loop design*: The elaborated goal models can be used for system design. Our objective of focusing on control loops is to extract highly independent modules from the requirements description; therefore, even though process control models, such as Shaw's [31], are still representative models for constructing control loops, the design and implementation of individual modules are not necessarily bound to these models. A developer can use several techniques to construct them. For example, the process proposed by Damas et al. [9] is useful for constructing behavior design models for individual modules from the goal model.

3) *Implementation*: We implement independent control loops. When change requests appear, we implement the identified changes in the source code of the corresponding control loops. If we extract additional control loops from the elaborated goal model, we implement them individually. Moreover, if the goals in the existing control loops are modified, we modify the source code of the corresponding control loops.

V. E M T

To evaluate our approach, we experimentally evolved[5] software in two different fields — a real-world modeling tool and a control system in a simulator. We compared our approach with baseline development styles through incremental software evolution. First, we evaluated the applicability of our approach and the accuracy of its impact analysis through a modeling tool evolution (Exp. 1). The target software was a KAOS modeling tool called the k-tool, which was developed by the National Institute of Informatics (NII). The k-tool has multiple views, including a main editor for drawing graphical diagrams and a tree view for observing all elements described in the model. The k-tool is implemented in Java and constructed on the basis of the *MVC (Model–View–Controller) model* [4], which is often used for GUI-based applications.

This experiment consisted of two parts. First, we constructed a goal model based on the elaboration process and implemented a tool that consisted of control loops extracted from the elaborated goal model. We also evolved this modeling tool according to evolution scenarios (Exp. 1-1). Next, we measured the accuracy of four examinees' impact analysis of each evolution by comparing the actual changes to the source code that we had implemented in Exp. 1-1 (Exp. 1-2).

We excluded some functions of the k-tool, all of which would be implemented in subsequent evolutions 1 to 3. We call the tool with the excluded functions the **k'-tool** and the tool that had the same functions as the k'-tool but was constructed by embedding control loops the **c-tool**. We sequentially evolved the k'-tool and c-tool by adding the three functions: *Evolution 1: Console.* We added a console with three features for displaying the modeling history (as a log), saving the log, and clearing it. *Evolution 2: Property view.* We added a property view in which we could define the properties of elements in the KAOS model. *Evolution 3: Uses definition tab.* We added a "Uses" tab to the property view to define the Uses labels for our elaboration process.

A. Exp. 1-1: Construction and Evolution

First, we implemented the c-tool on the basis of the elaboration of the goal model. We constructed c-tool by changing the k-tool implementation according to the extracted control loops. We extracted ten control loops as a result of the elaboration illustrated in Figure 8, five for managing individual views, four for providing common actions, and one for managing the KAOS model data. We constructed the c-tool by implementing classes that realized the individual control loops.

Next, we sequentially evolved the c-tool and k'-tool according to the evolution scenarios (Evolutions 1 to 3). While we evolved the c-tool according to the process described in Section IV, the k'-tool was evolved on the basis of a generic development process: first, we added new requirements to the goal model, then identified changes in the design model, and finally implemented these changes in the program code.

We compared the models of the two development processes after the evolution. Table I lists the characteristics of goal models in the k'-tool and c-tool development. Each row in the table represents the number of elements in the goal model. This table shows that the number of goals and entities increased as a result of the elaboration. The increase in goals and entities was due to the construction of generalized goals, additional collect-type goals, and the definition of process variables to embed control loops during the elaboration. On the other hand, this table also indicates that k'-tool and c-tool had nearly the same number of *concerns* links. The main reason is the aggregation of the sources of these links into goals located in the suitable control loops. As for the description of new requirements, while the new requirements in the goal model of the k'-tool were described in single clusters, those in the c-tool were decomposed and separately deployed for the purpose of adding or updating corresponding control loops. We also found that Evolutions 1 and 2 each added a control loop to the goal model of the c-tool.

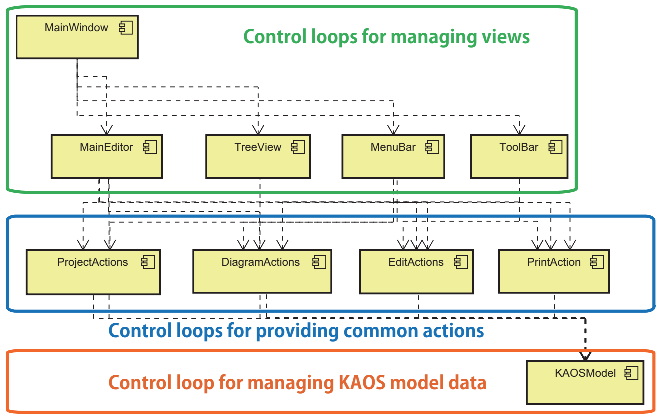
Table II shows the impact of the changes on the program code. We found that all of three evolutions of the k'-tool impacted all elements of the MVC model, i.e., the Model, View, and Controller. It increased the number of modified packages (*Mod* of *#Packages* in Table II). The MVC model isolates the domain model (Model) from the changes made to the user interface (View); however, since Evolutions 2 and 3 require the *Model* to be changed, changes had to be made to all elements of the MVC model. On the other hand, the changes to the c-tool were localized in the corresponding control loops. Table II also demonstrates that although the numbers of added or modified lines (ΔLOC) were almost the same in each development, more classes were modified

[5]Note that there still exists the possibility of a bias in coding and measurement in such experiments.

TABLE I

C ’- - .

		k’-tool (Elaboration NOT applied)				c-tool (Elaboration applied)			
		Initial	**Evol. 1**	**Evol. 2**	**Evol. 3**	**Initial**	**Evol. 1**	**Evol. 2**	**Evol. 3**
(i)	# Goals	120	124	143	144	168	172	198	200
(ii)	# Entities	15	17	19	19	28	31	34	34
(iii)	# Concerns	39	42	44	44	40	43	47	47
(v)	# CLs	–	–	–	–	10	11	12	12



Control loops for managing views
MainWindow
MainEditor
TreeView
MenuBar
ToolBar
ProjectActions
DiagramActions
EditActions
PrintAction
Control loops for providing common actions
Control loop for managing KAOS model data
KAOSModel

Fig. 8. The control loop hierarchy of the c-tool. Components represent individual control loops. Dashed arrows represent the uses dependencies.

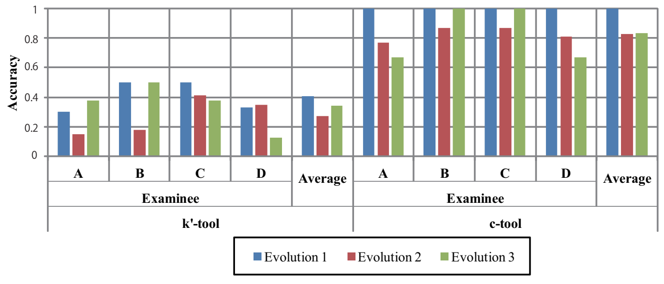


Fig. 9. Accuracy of impact analysis in Exp. 1-2.

TABLE II

C . “#P ”

.

		#Packages		#Classes		
		Mod	**Add**	**Mod**	**Add**	**ΔLOC**
Initial Dev.	k’-tool	–	7	–	263	56,668
	c-tool	–	10	–	278	57,015
Evolution 1	k’-tool	3	0	6	0	262
	c-tool	1	1	1	1	277
Evolution 2	k’-tool	4	0	26	8	7,873
	c-tool	2	1	23	8	7,864
Evolution 3	k’-tool	4	0	6	2	535
	c-tool	2	0	4	2	531

in the k’-tool development than in the c-tool development. The reason for this is that we added new classes for new control loops instead of modifying existing classes in the c-tool development. Adding a new module is generally easier than modifying an existing module; as a result, such an evolution has the potential to make the change implementation easier.

B. Exp. 1-2: Impact Analysis

Finally, four examinees, all of whom were graduate students in computer science and had experience in system development, carried out an impact analysis. The exercise used two kinds of documents, i.e., *goal models* as requirements documents and *class diagrams* as design documents of the k’-tool and c-tool, respectively. We asked the examinees to describe the requirements changes in the goal models and point out the classes that should be added or modified in the class diagrams in Exp. 1-1. We used recall to measure the accuracy of this impact analysis as follows: let C be the classes to be added or modified and P be the classes pointed out by the examinees. Thus, $accuracy = |C \cap P|/|C|$. Note that we have to consider the flexibility of the design. There is room to accommodate differences in how many classes should be added in the evolution. We accommodated such differences if the examinees correctly pointed out (1) the functions that the classes should provide, and (2) what package/control loop the classes should be injected into. On the other hand, we required a correct answer for the classes to be modified because the class diagrams provided enough information to identify these classes.

The results of this evaluation are plotted in Figure 9. We can see that all examinees ($A \sim D$ in Figure 9) were more accurate when they analyzed the c-tool than when they analyzed the k’-tool. One reason for this outcome was that there were fewer added and modified classes in the c-tool. Another reason resulted from whether examinees were able to detect changes in the *Model* element of the MVC model. The classes to be modified in the k’-tool were complexly spread over the elements of the MVC model. All of the examinees pointed out most of the changes for updating views and adding new operations to their impact analysis of the k’-tool, i.e., the changes in the *View* and *Controller*. However, they hardly ever pointed out changes in the *Model*. As for the c-tool, on the other hand, since the goals associated with the control loop for managing the model data were explicitly modified in the elaborated goal model, the examinees were able to find these changes.

VI. E C S

Next, we evaluated the changes in the code complexity of a cleaning robot on a simulator (Exp. 2), which we assumed

TABLE III

R . "#CL " . L "M:" "A:" #ΔC

, .

Ver	Additional functions	Baseline				Proposed style			
		#CLs	#ΔClasses	CBO	LOC	#CLs	#ΔClasses	CBO	LOC
1	Initial development	1	– , A:6	11	562	2	– , A:7	10	537
2	Battery maintenance	1	M:2, A:3	13	711	3	M:3, A:3	12	733
3	Additional cleaning method (Picking up)	1	M:2, A:1	15	818	3	M:1, A:2	12	836
4	Load management, unloading dust	1	M:4, A:4	18	1,106	5	M:3, A:7	12	1,256
5	Additional cleaning method (Wiping)	1	M:1, A:1	19	1,150	5	M:1, A:1	12	1,300
6	Obstacle avoidance	1	M:3	20	1,383	5	M:2	12	1,503
7	Combine multiple cleaning methods	1	M:4	20	1,450	5	M:4	12	1,557

to be a control system, through several evolution scenarios. We incrementally evolved a cleaning robot on the simulator to give it new functions, such as load management, obstacle avoidance, and additional cleaning methods. We compared our approach with a baseline development style, where one centralized component controlled other components in order to provide the services of the cleaning robot. We used two metrics for measuring the increase in code complexity; one is CBO (Coupling Between Objects) in CK metrics [7], which is the number of other classes to which it is coupled, and the other is McCabe's cyclomatic complexity [20], known as a metric for measuring the complexity of the software's control flow.

The experimental results are listed in Table III and plotted in Figure 10. The results in Table III show that the proposed style needed more additional code (LOC) than the baseline style when control loops were added (Ver 2 and Ver 4). The baseline style, on the other hand, rapidly increases the maximum number of CBOs, which represents the coupling complexity. This is because almost every evolution in this experiment requires changes to the centralized main method to deal with new classes that have been newly defined to satisfy new requirements, while the proposed style selects suitable control loops to couple with new classes.

The results plotted in Figure 10 show similar characteristics. The cyclomatic complexity represents the complexity of a software module, whose value increases according to the increase in decision points. The results show that the maximum cyclomatic complexity, which is the value for the method that carries out centralized decisions, in the baseline style increases after every evolution. Centralized controls tend to have enormous numbers of condition statements for identifying the current states of the system and its environment. The value of the proposed method in Figure 10, on the other hand, stays low because the changes in the code for each evolution are not concentrated and methods whose cyclomatic complexities are maximal are not always the same for each version of the robot.

VII. D

We will now discuss the development process in the light of our experiments.

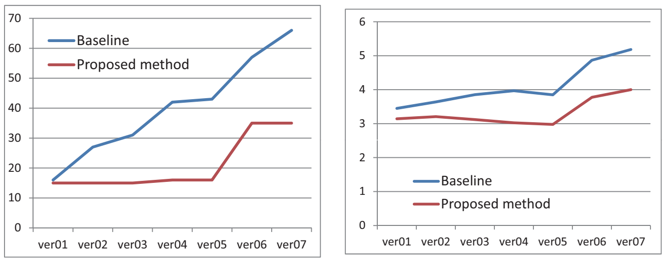


Fig. 10. Cyclomatic complexity: (Left) Maximum. (Right) Average.

A. Impact Analysis

As indicated in the results of Exp. 1, the evolution of a GUI-based MVC application tends to cause extensive changes in all *Model, View,* and *Controller* elements. The results also reveal that the elaborated goal model helped the examinees identify where changes should be implemented in the code, while it was difficult to detect changes by using the unelaborated goal model (for k'-tool). New requirements are generally described by adding new goals or modifying existing goals in the goal model. In our elaborated goal model, these goals are divided up if necessary and explicitly assigned to individual control loops. As a result, the elaboration localizes the impact of a change in the relevant control loops. New prime goals correspond to adding control loops, and changes corresponding to the descendant goals of existing prime goals correspond to the behavioral changes of their control loops. Note that we need to pay attention to the dependencies between control loops. If descendant goals of a generalized goal change, we have to verify whether the control loops dependant on the control loop associated with the generalized goal should be modified.

Our elaboration localizes the impact of changes in the relevant control loops; however, developers still need to identify which classes, methods, and attributes should be changed in these control loops. This is one of the limitations of our current elaboration process. Our elaboration process should also provide guidelines to prevent control loops from unnecessarily enlarging so that the impact analysis will be more accurate.

B. Change Implementation

The results obtained from Exp. 2 demonstrate that our development style reduces the modification costs and code complexity throughout the process of evolution. As Bhattacharya et al. recently validated their hypothesis that "*modules with higher modularity have lower associated maintenance effort*" [3], constructing a system with multiple control loops will localize the impact of a change, and as a result, it will prevent the complexity of the code and cost of the modifications from increasing.

As demonstrated in Exps. 1 and 2, we added control loops instead of modifying existing code in some of the evolution scenarios. Such an evolution would not only reduce the cost of modifications but also make the changes easier. In order to make a successful change, developers may have to focus on the granularity of their control loops. If developers embed goals for providing multiple functions into a single control loop, changes in the control loop may affect other functions even though the changes are confined to the control loop.

C. Applicability

The effectiveness of our elaboration depends on the size of the control loops in the goal model. If the extracted control loops are large, the impact of a change may spread throughout the corresponding control loops, and make it more difficult to identify the changes that have to be made to the program code. Our elaboration is beneficial when individual control loops are assigned single functions. Systems that have various input variables but whose actions individually depend on their input variables allow such assignments. For example, many GUI-based applications and web applications have such characteristics, because these applications have multiple widgets that tend to have their own input variables and independent actions. Control systems and applications in pervasive computing also seem to be suitable target applications for our elaboration, because these applications often deal with various input variables in their environments and have multiple actions to react to individual situations. On the other hand, applications based on the database-centric architecture, such as accounting software, tend to enlarge control loops for data management in the goal model; this is caused by aggregating concerns links into the entities associated with the database. We need to develop a new approach to supporting the evolution of database systems.

As for scalability, Table I for Exp. 1 suggests that the elaboration process consolidates concerns links in entities. Some studies have focused on the scalability of goal models in terms of the visibility [21]. Indeed, our elaboration focusing on similar goal generalization and modularization by embedding control loops enhances such scalability of goal modeling.

VIII. R W

There have been a number of studies on goal models that establish traceability links between the requirements and the system design. Lamsweerde [34] provided guidelines and heuristics on designing an architectural draft by deriving an abstract data flow of a system from the KAOS model with its formal specifications. This technique assigned the responsibility for achieving goals to the corresponding components and determined the connections among components from the data flows among goals. Yu et al. [37] proposed a technique for generating a highly versatile software design from the goal model. Their technique transforms goals into components and determines component connections from AND/OR-refinement links. This technique is effective for determining a set of components that take responsibilities for achieving goals. While these approaches establish traceability, our goal model elaboration mainly aims to limit the impact of changes by applying the control loop pattern that provides a criteria for determining the component's responsibility.

Some studies have devised ways to support developers in making requirements changes in the goal model. Ernst et al. [12] uses a knowledge base to find advantageous solutions to requirements changes while minimizing the effort required to implement the new solutions. This approach uses the goal model to find new operations that are additionally required. Our elaboration process, on the other hand, allows for the impact analysis on implementation artifacts once requirements changes have been identified. Cleland-Huang et al. [8] proposed a probabilistic approach to managing traceability links for non-functional requirements. This technique helps designers to analyze the impact of changes by retrieving links between classes affected by changes in a softgoal interdependency graph. Our approach analyzes the impact of changes by checking changes in relevant control loops and their dependencies. While Cleland-Huang et al. use the interdependence of non-functional requirements, our approach is based on the structure of the functional requirements and their *concerns* links to entities.

There have been many studies on requirements engineering that deal with the activities of control loops. Salifu et al. [28] proposed an approach to specifying monitoring and switching behaviors by using problem frames notations. Lapouchnian et al. [17] used a goal-oriented description to acquire all possible behaviors and determine which behavior is the most appropriate at any given moment. Landtsheer et al. [11] presented a technique of deriving event-based specifications for control software, written in tabular language, from the KAOS description. Cheng et al. [6] provided a goal-based modeling approach to describing requirements for dynamically adaptive systems. Wang and Mylopoulos [36] used a goal model to define monitoring, diagnostic, and reconfiguration rules to add high variability to a legacy software system. Chen et al. [5] proposed a self-tuning method for the survivability assurance of Web systems based on the goal model. Their method uses a control loop and determines the configuration by referring to the goal model with annotated contribution values. While these approaches describe activities concerned with control loops governing a system or for making adaptive behaviors, such as those for constructing self-adaptive systems [29], our approach aims at identifying independent components from the goal model so as not to spread the impact of changes to other components when systems evolve.

IX. C

We have defined a goal model elaboration process to deal with software evolution. This elaboration process aims at defining control loops in the goal model and extracting them as system components. An elaborated goal model can localize changes in relevant control loops, and this helps to identify where we should modify our source code and analyze the impact of changes. The results of experimental evolutions of a modeling tool and control software suggested that our elaboration process can provide highly accurate impact analysis and suppress any increases in the complexity of the program code by extracting independent components from the goal model that allow us to localize changes. One of our on-going studies pertains to further support for identifying behavioral changes by introducing formal goal specifications. We believe that such a formal approach can help us to perform accurate impact analyses and change implementations.

R

[1] A. April and A. Abran. *Software Maintenance Management: Evaluation and Continuous Improvement*. IEEE CS, 2008.

[2] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–94, Nov. 1993.

[3] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In *Proc. of the 34th International Conference on Software Engineering (ICSE '12)*, pages 419–429. IEEE, 2012.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.

[5] B. Chen, X. Peng, Y. Yu, and W. Zhao. Are your sites down? requirements-driven self-tuning for the survivability of web systems. In *Proc. of the 19th IEEE International Requirements Engineering Conference (RE '11)*, pages 219–228, 2011.

[6] B. H. C. Cheng, P. Sawyer, N. Bencomo, and J. Whittle. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In *Proc. of the 12th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS '09)*, pages 468–483. Springer, 2009.

[7] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[8] J. Cleland-Huang, R. Settimi, O. BenKhadra, E. Berezhanskaya, and S. Christina. Goal-centric traceability for managing non-functional requirements. In *Proc. of the 27th international conference on Software engineering (ICSE '05)*, ICSE '05, pages 362–371. ACM, 2005.

[9] C. Damas, B. Lambeau, and A. van Lamsweerde. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *Proc. of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (SIGSOFT '06/FSE-14)*, pages 197–207. ACM, 2006.

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

[11] R. De Landtsheer, E. Letier, and A. van Lamsweerde. Deriving tabular event-based specifications from goal-oriented requirements models. In *Proc. of the 11th IEEE International Conference on Requirements Engineering (RE '03)*, pages 200–210. IEEE CS, 2003.

[12] N. A. Ernst, A. Borgida, and I. Jureta. Finding incremental solutions for evolving requirements. In *Proc. of the 19th IEEE International Requirements Engineering Conference (RE '11)*, pages 15–24. IEEE CS, 2011.

[13] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. Integrated impact analysis for managing software changes. In *Proc. of the 34th International Conference on Software Engineering (ICSE '12)*, pages 430–440. IEEE, 2012.

[14] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proc. of the First International Conference on Requirements Engineering (RE '94)*, pages 94–101. IEEE CS, 1994.

[15] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proc. of the 31st International Conference on Software Engineering (ICSE'09)*, pages 78–88. IEEE CS, 2009.

[16] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[17] A. Lapouchnian, Y. Yu, S. Liaskos, and J. Mylopoulos. Requirements-driven design of autonomic application software. In *Proc. of the 2006 conference of the Center for Advanced Studies on Collaborative research (CASCON '06)*, pages 80–94, 2006.

[18] M. M. Lehman and J. F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15–44, 2001.

[19] E. Letier. *Reasoning about Agents in Goal-Oriented Requirements Engineering*. PhD thesis, Universite Catholique de Louvain, 2001.

[20] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308 – 320, Dec. 1976.

[21] G. Mussbacher, D. Amyot, J. Araujo, A. Moreira, and M. Weiss. Visualizing aspect-oriented goal models with AoGRL. In *Proc. of Second International Workshop on Requirements Engineering Visualization (REV '07)*, pages 1 – 10, 2007.

[22] H. Nakagawa, A. Ohsuga, and S. Honiden. gocc: A configuration compiler for self-adaptive systems using goal-oriented requirements description. In *Proc. of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '11)*, pages 40–49. ACM, 2011.

[23] B. Nuseibeh. Weaving together requirements and architectures. *IEEE Computer*, 34(3):115–117, 2001.

[24] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proc. of the 26th International Conference on Software Engineering (ICSE '04)*, pages 491–500. IEEE CS, 2004.

[25] D. L. Parnas. Software aging. In *Proc. of the 16th international conference on Software engineering (ICSE '94)*, pages 279–287. IEEE CS, 1994.

[26] V. Rajlich and J. a. H. Silva. Evolution and reuse of orthogonal architecture. *IEEE Trans. Softw. Eng.*, 22(2):153–157, Feb. 1996.

[27] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27:58–93, January 2001.

[28] M. Salifu, Y. Yu, and B. Nuseibeh. Specifying monitoring and switching problems in context. In *Proc. of the 15th IEEE International Requirements Engineering Conference (RE '07)*, pages 211–220. IEEE CS, 2007.

[29] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein. Requirements-aware systems: A research agenda for RE for self-adaptive systems. In *Proc. of the 18th IEEE International Requirements Engineering Conference (RE '10)*, pages 95–103. IEEE CS, 2010.

[30] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley Professional, 2003.

[31] M. Shaw. Beyond objects: a software design paradigm based on process control. *SIGSOFT Software Engineering Notes*, 20:27–38, January 1995.

[32] I. Sommerville. *Software engineering 8*. Addison-Wesley, 2007.

[33] A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proc. of the Fifth IEEE International Symposium on Requirements Engineering (RE'01)*, pages 249–262. IEEE CS, 2001.

[34] A. van Lamsweerde. From system goals to software architecture. In *Proc. of Formal Methods for Software Architectures*, volume LNCS 2804, pages 25–43. Springer, 2003.

[35] A. van Lamsweerde, E. Letier, and R. Darimont. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering*, 24(11):908–926, 1998.

[36] Y. Wang and J. Mylopoulos. Self-repair through reconfiguration: A requirements engineering approach. In *Proc. of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*, pages 257–268. IEEE CS, 2009.

[37] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos, and J. C. S. P. Leite. From goals to high-variability software design. In *Proc. of the 17th international conference on Foundations of intelligent systems (ISMIS'08)*, pages 1–16. Springer-Verlag, 2008.