Modeling Articulation Work in Software Engineering Processes

Peiwei Mi[†] and Walt Scacchi[‡]
Computer Science Department[†] and Decision Systems Department [‡]
University of Southern California[1], Los Angeles, CA 90089

Abstract

Current software process modeling techniques do not generally support *articulation work*. Articulation work is the diagnosis, recovery and resumption of development activities that unexpectedly fail. It is an integral part of software process enactment since software processes can sometimes fail or breakdown. This paper presents a knowledge-based model of articulation work in software engineering processes. It uses empirically-grounded heuristics to address three problems in articulation work: diagnosing failed development activities, determining appropriate recovery, and resuming software processes. We first investigate the role and importance of articulation work with respect to planned software development activities. We then outline a knowledge-based model of articulation work. The model has been implemented in a knowledge-based software process modeling environment called the Articulator. Combining the available software process modeling techniques and the model of articulation leads to a better foundation in process improvement and evolution.

1 Introduction

Software process modeling is aimed at understanding and facilitating software development processes. A *software process model (SPM)* is an abstract representation of a software engineering process [HK89, MS90, Ost87]. Enacting an SPM enables software developers to carry out development according to the activities specified in the SPM [Dow90, MS91b]. SPMs act as plans that anticipate types of development activities involved, their performance sequence, and resources needed to perform them. To this end, development activities specified in SPMs are articulated or planned activities, and software development under the guidance of an SPM becomes an organized execution of the SPM.

While SPMs provide powerful assistance, they fail at handling unexpected events that interrupt software development. Such an interruption is called a *breakdown* of planned development activities. A number of empirical studies of development work have revealed both the frequent occurrence of such breakdowns, and a recurring type of activity that resolves or mitigate them [BS87, BS89, CKSI87, CKI88, KCI87, Gas86, GS86, GKC87, Str88, WEKC87]. This type of resolution or recovery activity is called *articulation work*, which normally occurs as an emerging response to breakdowns in planned activities and is difficult to describe in existing SPMs.

Articulation work is an integral part of SPM enactment because of the inherent existence of breakdowns. Due to the unpredictability of changes in the real world, unexpected events happen during software development. Similarly, since our knowledge of how software processes are actually enacted is limited, mistakes or oversights are made in configuring activities and resources in SPMs.

[1]Acknowledgements: This work has been supported in part by contracts and grants from AT&T, Northrop Inc., Pacific Bell, and the Office of Naval Technology through the Naval Ocean Systems Center. No endorsement implied.

Recommended by: Bill Curtis

0-8186-2490-6/91/0000/0188$01.00 © 1991 IEEE

Subsequently, articulation work is the response developers make to adapt to changes in the real world and to correct previous misjudgements.

Articulation work is also a complementary technique to process modeling. Adaptation of an existing SPM is a common approach to create a customized SPM. However, practical experience can help evolve SPMs to more matured forms that better address process situations that developers frequently encounter, or where more automated support is required. As an SPM is enacted, articulation work effectively updates or augments the SPM to better fit the local environment and expands it in overlooked aspects. The continuous application of articulation work results in a better situated SPM that can be more successfully enacted in the future.

We present a strategy for modeling and supporting articulation work in this paper. This strategy focuses on constructing a knowledge-based model of articulation work based on empirical studies of various kinds of development work. It abstracts knowledge and skill of articulation work from these studies and formalizes them into a *computational process* of articulation and *articulation heuristics* that direct the result of the articulation process. In turn, we describe two types of articulation heuristics: *problem-solving heuristics* to indicate plausible solutions to breakdowns, and *selection heuristics* to constrain the use of these solutions according to circumstances that surround a breakdown.

In what follows, we identify a set of important features of process modeling from some representative software process modeling techniques in Section 2. In the same section, we also present findings from representative empirical studies on software development and other types of work. They help to reveal the oftentimes circumstantial and idiosyncratic practices of software engineering. In Section 3, we present the results of our previous work to date in the study of software engineering processes with a process modeling environment called the Articulator[MS90]. In Section 4, we present a knowledge-based model of articulation work in terms of a process description of articulation work and a set of articulation heuristics. We conclude the paper by summarizing some major benefits provided by the model of articulation work.

2 Related Work

In this section, we first summarize some representative techniques for modeling software processes and identify a group of common capabilities that are used to model planned development activities. Then, we review several empirical studies that investigate the practice of software development in different settings. In these studies, articulation work is repeatedly observed, and it is found to be a type of software development activity that is unsupported. By comparing the current process modeling techniques and the empirical studies, we identify articulation work as an area that on the one hand is critical to the success of software development, but on the other hand lacks of support from these modeling techniques.

2.1 Modeling Techniques for Software Processes

Modeling software processes requires abstracting important features of software development activities, formalizing them into SPMs, and supporting development through enactment of the SPMs. This section surveys a group of modeling techniques and prototype systems that are representative of state-of-the-art technology. We group these modeling techniques into four categories based the modeling formalisms: Language-based modeling, such as Arcadia [TBCO89]; Rule-based modeling, such as Marvel [Kai88]; AI-based modeling, such as Grapple [HL88]; Object-based modeling, such as IPSE 2.5 [War89]; and Evolution-based modeling, such as Prism[MG90] and TAME[BR88].

We characterize the modeling techniques by the type of development activities they model, the modeling formalism, the elements in a model, and the tools supporting process modeling:

1. SPMs created by these modeling techniques consist of descriptions of development activities and resources. Different techniques have their own variations of SPMs. Most provide a specification language or notation to let users describe SPMs. Some modeling techniques also provide a meta-model.[2] For example, Marvel uses inference rules to model software development, while Grapple uses plans. Other techniques use object-oriented or object-based specification. Prism has a Petri-net based meta-model of SPMs.

2. Development activities are a major part of an SPM. They are an abstract description of planned development activities. A description of planned development activities includes activity decomposition, execution sequence, or resource usage. The activities are represented either as inference rules (Marvel), plans (Grapple), and programming language constructors (Arcadia).

3. Enactment of SPMs guides software development by indicating active or suspended activities and by invoking needed tools. Software development is carried out according to the sequence of planned activities in an SPM. This performance sequence can be explicit as in Arcadia process programs, or implicit as effects of production rules in Marvel and subgoals in Grapple.

4. Prism and TAME are focused on analyzing and improving SPMs through empirical measurements, evaluations and process refinements. They suggest a learning and feedback process so that developers can improve their practice of software engineering by having a better SPM. However, at this time, process improvement is mainly performed off-line.

5. None of the modeling techniques provides a mechanism describing and modeling unexpected breakdowns in planned activities. Enactment is only responsible for carrying out SPMs. When a breakdown occurs, enactment will be interrupted and development stopped. The enactment and other support of the SPM will be suspended until some external remedy, or intervention occurs.

Although not modeled in these SPMs, articulation work nonetheless exists in software development. It has been repeatedly identified as a crucial part of development work as shown next. As such, we will now review a set of empirical studies that report and substantiate the repetitive occurrence of articulation work.

2.2 Related Empirical Studies

A number of empirical studies have examined the organization and behavior of people in software development projects and other types of development work. In these studies, two types of activities are identified. One type of activity is well understood, mostly routine, and performed with expected order. For example, developers often plan to construct programs according to a previously prepared architectural design. Articulation work, however, is a response to emergent situations that unexpectedly cause planned activities to breakdown. For instance, a compiler bug may be discovered while writing and compiling application programs. The architectural design and programming activities should anticipate application program debugging, but generally would not anticipate compiler bugs. However, the programmers involved must determine what to do in order

[2] A software process meta-model is a formalism used to describe a family of SPMs.

Researcher	*Development Work*	*Form of Breakdown*	*Type of Breakdown*	*Artic. Method*
Bendifallah & Scacchi [BS87]	Software Maint.	Contingencies of Primary Work	Changes in Workplace	Accommodation and Negotiation
Bendifallah $ Scacchi [BS89]	Software Spec	Contingencies of Primary Work	Idiosyncratic Setup, Changes in Workplace, Work Structure Adequacy	Negotiation, Shifts in Work Structure
Krasner, Curtis & Iscoe [KCI87]	Software Design and Communication	No Communication, Miscommunication, Conflicting Information	Difference in Skill, Incentive, Changes in Workplace, Other Breakdowns,	Boundary Spanner as Articulator
Guindon, etc [GKC87]	Software Design Group Work	Ineffectiveness and Difficulties in Work	Lack of Knowledge, Cognitive Limitation	N/A N/A
Walz, etc [WEKC87]	Req's Definition, Group Work	Contingencies of Group Work	Lack of Knowledge, Group Interaction	N/A N/A
Gasser [Gas86]	Routine Use and Evolution of Organizational Sys.	Disruption and Contingency of Routine Work	Misfit of Resources	Fitting, Augmenting, Workaround
Gerson & Star [GS86]	Routine Use and Evolution of Office Systems	Conflict of Interests	Different Viewpoints, Changes in Workplace	Skepticism, Trade-off, A special articulator
Suchman [Suc83]	Routine Use of Office Equipment	Equipment malfunction or becomes inoperable	People do not follow operation procedures	Interaction, Negotiation
Strauss [Str88]	General Project Work	Disruption, Contingencies	Nonroutineness, Altered Arrangements	Interaction, Alignment, Negotiation

Figure 1: Summary of The Empirical Studies

to eventually complete the programming activity, either by getting the compiler bug fixed, or by somehow working around it. Thus, our focus of attention in reviewing these studies is to identify and compare what kinds of breakdowns occur, and what articulation work is performed in response. We have collected three representative groups of empirical studies: those from the USC System Factory project[BS87, BS89], MCC's Design Process Group[CKSI87, CKI88, GKC87, KCI87, WEKC87], and other studies of system development work[Gas86, GS86, Suc83, Str88].

Figure 1 lists the basic findings about articulation work in these empirical studies in terms of types of development work examined, forms of process breakdowns that occur, reported types of breakdowns, and the methods of articulation work observed in response. From these findings, we draw several conclusions:

1. Software development and other types of systems work consist of two types of activities: planned activities and articulation work. Planned activities which account for expected work flow constitute the routine or conventional part of software development. Articulation work, on the other hand, is the normal response to breakdowns in planned activities. The purpose of articulation work is to restore or reorganize intended work flow. The occurrence of breakdowns is a recurring phenomenon developers experience in many types of work. In software development, breakdowns have been observed in most of the development phases, and likely occur throughout the entire system life cycle. Further, coping skills and experience in similar situations seem to provide a behavioral foundation for articulation work.[3]

2. Articulation work mainly involves three kinds of activities: diagnosing breakdowns, searching

[3]This suggests that the ability or inability to call upon such skills and experiences can help distinguish competent and naive articulation work effort.

for solutions, and implementing a suggested solution. The relationship between planned activities and articulation work is identified as an interruption-driven interaction.

3. Seven major types of breakdowns are reported: 1) Some activities are left unfinished; 2) Unnecessary activities have been done; 3) Developers have to use other resources than allocated; 4) Developers do not get necessary resources; 5) Developers get different resources than they expect; 6) Developers can not use assigned resources that are being used by others; 7) Resources are unavailable when they are needed. These types are abstracted forms of those types identified in Figure 1.

4. Several articulation methods are also abstracted from Figure 1 as ways developers follow when performing articulation work. They represent effective articulation skill gained through experience. In our system, we consider six major types of methods: 1) replace a resource with another of the same class; 2) replace a resource with another of a similar class; 3) restructure an SPM, such as add, delete activities and relations among activities; 4) modify values of resources; 5) redo some activities; 6) split or merge activities.

After we compare these findings with the process modeling techniques, it becomes apparent that current software process modeling techniques do not directly address articulation work. First, the nature of articulation work has not been understood and modeled from a software process viewpoint. Second and more important, the situated occurrence of articulation work is unpredictable though potentially frequent in a development project. Articulation work *can not be planned* per se in an SPM. Therefore, we believe a new modeling technique is needed. To this end, these empirical studies provide an empirically-grounded foundation for a model of articulation. The model of articulation outlined in this paper is thus based on this foundation.

3 The Articulator: An Environment for Studying Software Processes

The Articulator is a knowledge-based environment that models and simulates software engineering processes. This section provides a brief overview of the Articulator, while the details appear elsewhere [MS90, MS91b, MS91a].

The Articulator provides a meta-model of software engineering processes and workplace. The simulator in the Articulator symbolically enacts SPMs in an artificially defined workplace. SPMs in the Articulator are described in an object-oriented process specification language that incorporates rule-based procedural and non-procedural methods [MS90].

The Articulator meta-model is a web of object classes. Each of the object classes represents a type of development resource useful in software development. Definition of an object class includes a set of attributes, relations with other object classes, and inheritance from its super-class. Some important object classes are: 1) individual developers that are modeled as *agents*[4] with problem-solving capabilities, 2) development organization agents with organizational structure and local hierarchy or authority, 3) development tasks with activity hierarchy, development tools, resource requirements, and resource possession, 4) workplaces with embedded resources, and 5) development tools, such as different kinds of hardware and software. Altogether, these foundational object classes that define the *class database* constitute the set of building elements for describing software engineering processes and workplaces.

[4] Agents represent a cluster of interrelated objects, attributes and behavioral methods.

In the Articulator, an SPM describes an development approach such as the waterfall model, the Spiral model, or some other formulation [Sca87]. An SPM is represented as an extension of the Articulator meta-model. The *model database* in the Articulator stores all defined SPMs. Definition of an SPM includes a set of subclasses of objects in the class database and a set of constraints in terms of defined attributes. Such a defined SPM is *prescriptive* because it is a generalization of a set of software processes and represents a projection of what should happen in them.

An instance of an SPM represents a process enactment in a particular workplace. In this sense, multiple enactments of a single SPM creates multiple instances for it. The Articulator supports two types of instantiation: simulation and development. A *simulation instance* is created in an artificial workplace by the Simulator and recorded in the Articulator. A *development instance* is a historical, empirically grounded record of real development activities that followed the SPM in a real workplace. Such records are entered through external interfaces[MS91b]. Instances of SPMs are *descriptive* since they are (replayable) records of what happened in a software process, either in an artificial or real situation. In the Articulator, the *simulation database* and the *development database* store simulation and development instances respectively. The Instantiation Manager in the Articulator provides cross reference among the SPMs and their individual instances.

Software development is simulated in the Articulator as the creation or reconfiguration of an instance of an SPM in a defined workplace. At first, the Simulator binds the SPM with a set of developers, tools and resources in a workplace according to the SPM's resource requirement when the required resources become available, the simulation then carries out the SPM with the initial resource possession. Then, during simulation, the modeled agents must accommodate circumstantial changes in the workplace, such as unexpected shifts in resource possession, which are to be resolved by articulation.

The Articulator is also able to simulate circumstantial changes which can cause the four major types of breakdown reported in Section 2. By using the Articulator, the types of breakdowns are modeled by different configurations of resources and their relations in specified situations to simulate the occurrence of breakdowns. For example, changes in the workplace are cited as a type of breakdown where the evolution of a resource makes it unavailable for use when it is needed. In the Articulator, workplace changes are represented by changes in the related objects and their associated values spanning over a period of time. For example, the operational viability of a computer can be represented by an attribute, called *status*, with values such as *available* or *unavailable* [MS91a]. In this case, switching from *available* to *unavailable* signals the computer is effectively out of service. Thus, some actions that use the computer at the moment may break down. This represents a circumstantial configuration of resources under which a particular type of breakdown occurs. Accordingly, we now turn to discuss a model of articulation that can resolve either artificially created breakdowns in process simulation or to help resolving real identified breakdowns during process enactment.

4 The Model of Articulation Work

In this section, we present a conceptual model of articulation work. The model is an abstraction of articulation work observed in the empirical studies. It describes a process of articulation driven by a set of articulation heuristics. At the end, we illustrate an example of the model of articulation.

4.1 Process of Articulation

When a breakdown occurs, the agent performing the activity that broke is responsible to initialize articulation work to create a solution to the breakdown. A *solution* is a set of replacing activities

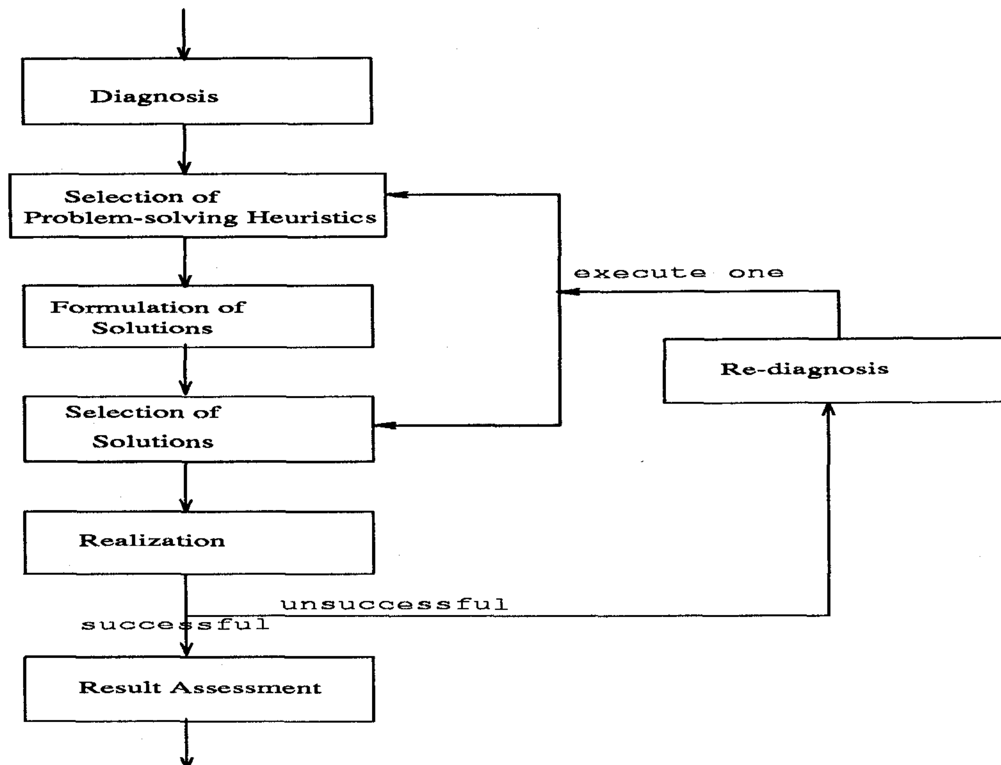

Figure 2: A Process of Articulation

or changes in resource possession. By executing the solution, either the broken activity can be recovered and resumed, or the identified problem can be solved else avoided. Articulation generally consists of the following stages: diagnosis of the breakdown, selection of problem-solving heuristics by selection heuristics, formulation of solutions by the problem-solving heuristics, selection of solutions by selection heuristics again, realization of the solution, rediagnosis and iteration if needed, and result assessment (Figure 2), which is implemented through inference rules. Figure 3 lists an example rule in the diagnosis stage. This rule detects a missing required resource and creates a schema to store this information. The three conditions for the rule state that an `individual agent` performs an `action`, which `requires` and `uses` resources. The conditions are satisfied if there is a resource that is required but not available for use. The assertions then direct the articulation process to select problem-solving heuristics (as goal `articulate-select-psh`) and create an object (by `cschema`) to save relevant information, such as the agent, the diagnosis, etc. Problem-solving and articulation heuristics are an abstraction of problem-solving and articulation methods identified before. They will be discussed in the next section.

In diagnosis, the agent identifies what type of breakdown occurred based on the symptoms of the breakdown and the available types of breakdowns. Next, based on its current set of problem-solving heuristics, the agent uses its selection heuristics to select a subset of problem-solving heuristics, which according to the selection heuristics satisfies the agent's current preference and its effectiveness in resolving the breakdown. Then, the agent applies the chosen problem-solving heuristic to create solutions. At this moment, one problem-solving heuristic may lead to several possible solutions. These solutions are evaluated according to the selection heuristics again in order to decide which one is most preferable. The selected solution is then executed to resolve the breakdown. If the breakdown is resolved successfully, articulation is complete. Otherwise, the agent needs to

```
;; diagnosis: a required resource is not available

(p resource-not-available :context t
  (individual-agent  ^schema-name <agent>
                     ^controller-goal articulate-diagnosis
                     ^status <action>)
  (action  ^schema-name <action>
           ^task-using-resource <reso-used>
           ^task-require-resource <reso-req>)
  (resource  ^schema-name <reso>
             ^schema-name (not-member <> <reso-used>)
             ^schema-name (member <> <reso-req>))
 -->
  (new-value <agent> 'controller-goal 'articulate-select-psh)
  (new-value <agent> 'controller-parameter
           (intern (string-append 'articulate+ (unique-name))))
  (cschema (get-value <agent> 'controller-parameter)
           ('instance 'articulate)
           ('performer <agent>)
           ('diagnosis 'missing-required-resource)
           ('broken-action <action>)
           ('needed-resource <reso>)))
```

Figure 3: A Rule of Detecting A Missing Resource

re-diagnose the breakdown. Based on the result of rediagnosis, the agent may repeat one of selected heuristics and selected solutions. For example, if the chosen problem-solving heuristic did not work well in the last solution, selection of heuristics must be redone in order to choose a new problem-solving heuristic. This subprocess is repeated until the breakdown is removed successfully and the task can be resumed following a modified SPM. In the last step, the realized solution and the changed SPM will be assessed to see if they need to be saved permanently into the model based in the Articulator. Alternatively, if in the end none of the available heuristics resolves the breakdown, the task is suspended until some other external intervention occurs and the agent moves on to another task.

4.2 Heuristics of Articulation

Knowledge of articulation work is represented in the form of heuristics. They are used to direct how articulation work is carried out, how a solution is formulated and selected, and how a solution is realized. But as heuristics, they can not guarantee that a complete solution to a breakdown can be found and applied. Nontheless, in the this section, articulation heuristics are discussed in more detail.

There are at least two types of articulation heuristics: *problem-solving heuristics* that direct types of solutions for breakdowns, and *selection heuristics* that direct selection of plausible solutions. Problem-solving heuristics represent knowledge of articulation methods that often resolve various breakdowns. The model of articulation utilizes problem-solving heuristics to create particular solutions to a breakdown. Although problem-solving heuristics are able to resolve breakdowns, they are not equally applicable to different breakdowns in different circumstances. Also, people (and agents) may prefer certain methods over others. Selection heuristics, therefore, represent constraints on the applicability of problem-solving heuristics. Two type of selection constraints

No.	*Heuristics*	*Possible Solutions*
1	Replace-instance	Find another compiler
2	Replace-class	Find alternative task to do the job, e.g. use another language
3	Restructure	Add a debug activity to fix the compiler or Add a write activity for a new compiler
4	Modify	n/a in this case
5	Redo	n/a in this case
6	Split/Merge	n/a in this case
7	Change Agent	Report to the team leader, or look for service from someone else
8	Create a new alternative subtask	Learn another language
9	Impose new constraints in the SPM	Check the complier before use
10	Others	Wait, Switch to another task

Figure 4: A Sample Set of of Problem-Solving Heuristics and An Example

are possible. *Global selection heuristics* define overall constraints or personal preferences over problem-solving heuristics and solutions. *Circumstantial selection heuristics* limit the applicability of problem-solving heuristics in various situations.

- *Problem-solving heuristics* describe articulation methods to resolve breakdowns. For different types of breakdowns, people use different methods. A set of effective articulation methods, therefore, gives a developer the capability to resolve a wide range of breakdowns. Our goal is to establish a set of *problem-solving heuristics* that indicate possible solutions for different types of recurring breakdowns. Figure 4 lists a sample set of problem-solving heuristics. It also lists possible solutions for our example of articulation work in Section 4.3, which will be explained later. These heuristics are an extension of articulation methods identified before. Figure 5 gives two inference rules that implement part of the problem-solving heuristic: search for an existing alternative action. The first rule identifies the currently selected problem-solving heuristic, `find an alternative task`. The second rule finds an alternative action in the agent's past experience, which lists tasks the agent has performed before. The alternative task is selected because it provides the same set of resources as the broken action, but requires different resources. The assertions replaces the broken action by the newly selected one and continues the articulation process.

- *Global selection heuristics* describe global constraints or personal preferences over problem-solving heuristics and solutions. They may exclude some under certain circumstances or just specify an order of selection. Figure 6 lists a sample set of global selection heuristics and their implications. Some of them may be mutually exclusive, while others may mutually inclusive. Global selection heuristics are used twice in the model of articulation. First, they are used to select problem-solving heuristics before solutions are formulated. In this case, some of the problem-solving heuristics are excluded because of the existence of some global selection heuristics in the current configuration of the model of articulation, others may be ordered. After that, the problem-solving heuristics on top of the ordered list will be applied to create possible solutions to a breakdown. Furthermore, the ordered list is subject to circumstantial heuristics as explained below. Second, global selection heuristics are used in selecting solutions once they are formulated. Here, the solutions may come from a single problem-solving heuristic or a group of them.

```
;; start articulation with heuristic: find an alternative task.

(p articulate-by-alt-task :context t
  (individual-agent  ^schema-name <agent>
                     ^controller-goal articulate-select
                     ^status <action>)
  (articulate  ^schema-name <action>
               ^currently-chosen-psh find-alt-task)
-->
  (format t "Articulate by find-alternative-task")
  (new-value <agent> 'controller-goal 'articulate-alt-task-select))

;; find an alternative task in the agent's experience

(p articulate-alt-task-select-find-in-experience :context t
  (individual-agent  ^schema-name <agent>
                     ^controller-goal articulate-alt-task-select
                     ^status <action>
                     ^individual-agent-has-experience <experience>)
  (experience  ^schema-name <experience>
               ^experience-has-task <new-action>)
  (articulate  ^schema-name <action>
               ^currently-chosen-psh find-alt-task
               ^symptom <<non-existing-resource non-authorized-resource>>
               ^broken-action <p-action>
               ^needed-resource <p-resource>)
  (action  ^schema-name <p-action>
           ^task-provide-resource <pro-reso>)
  (action  ^schema-name <new-action>
           ^status <> selected
           ^task-provide-resource (set-equal <> <pro-reso>)
           ^task-require-resource (not-member <p-resource> <>))
-->
  (add-alternative-list <agent> <p-action> <new-action> <p-resource>)
  (new-value <new-action> 'status 'selected))
```

Figure 5: Portion of Rules for Problem-solving Heuristic No. 2

- *Circumstantial selection heuristics* describe an order of selection of problem-solving heuristics for each of the identified types of breakdowns. They may also include other local constraints. For example, use of some resources can be prohibited at a given moment. Circumstantial selection heuristics basically indicate under what circumstances which heuristics are more effective and therefore more preferable. Our current study suggests a possible ordering of problem-solving heuristics for different types of breakdowns (Figure 7), where the numbers refer to the problem-solving heuristic defined in Figure 4. From Figure 7, it is easy to observe that some of problem-solving heuristics are not applicable in certain circumstances. Also some problem-solving heuristics in Figure 7 are grouped to indicate they are equally applicable in the situation. For example, like global selection heuristics, circumstantial selection heuristics help to select and order problem-solving heuristics before they are used to find solutions for a breakdown. However, circumstantial selection heuristics differ from global selection heuristics in that they depend heavily on the context of development situations and may change dynamically. Otherwise, use of circumstantial selection heuristics is similar to that of

No.	Global Selection Heuristics	Solution Selection
1	Find a solution	One solution is enough
2	Find no solution	No solution is searched
3	Minimize extra tasks	Solutions with less tasks
4	Maximize extra tasks	Solutions with more tasks
5	Minimize tasks by self	Solutions with less tasks by self
6	Maximize tasks by self	Solutions with more tasks by self
7	Minimize tasks by others	Solutions with less tasks by others
8	Maximize tasks by others	Solutions with more tasks by others

Figure 6: A Sample Set of Global Selection Heuristics

Types of Breakdowns	Ordering of PS Heuristics
Not Finished Activity	Restructure, Modify, Split
Unnecessary Activity	Restructure, Modify
No Resource Specification	Modify, Restructure
No Resource Allocation	Replace-instance, Replace-class, Restructure, Redo
Unavailability of Resource, occupied	Replace-instance, Replace-class, Restructure, Redo
Unavailability of Resource, broken	Replace-instance, Replace-class, Restructure, Redo
Others	All of them can be used

Figure 7: A Sample Set of Circumstantial Selection Heuristics

global selection heuristics.

We would like to emphasize here that these heuristics are not definite solutions for breakdowns. In some cases, they may not be applicable. However, they do represent experiences that expert developers often use. Furthermore, the more such heuristics a developer has, the more flexibility she has in articulation work.

4.3 An Example of Articulation

As an example of how the model of articulation works, consider the following: A team of programmers is involved in developing a graph editor [KS90]. During development, the team follows a customized waterfall development process model (let us name it `Model W`), where all development stages, milestones, tools, and used resources are specified. Major development stages in `Model W` include requirements definition, architectural design, detailed design, implementation, test, and integration. The project is started as expected. Everything proceeds according to `Model W` until a compiler bug is discovered while a junior programmer compiles her programs. The architectural design and implementation activities generally would not anticipate such compiler bugs. However, the involved junior programmer, and the team as well, must determine what to do in order to eventually complete the overall development activity. Therefore, articulation work is started by the junior programmer, which in turn initiates the model of articulation work.

The first articulation activity is to evaluate the circumstances and to identify the type of the breakdown. In this situation, the compiler with the bug is assigned as a tool for implementation task. It turns out that this is a *Unavailability of Resource-broken* because the compiler could not be successfully applied due to the existence of bugs.

There exist a number of possible solutions for this breakdown as indicated by the available problem-solving heuristics (see Column 3 in Figure 4). The programmer may stop her activity without doing anything (Heuristic 10.1), or she may switch to another task if she has one (Heuristic 10.2). These two methods involve only schedule changes without solving the problem. Taking more

active approach, the programmer may search for alternatives to replace the problematic compiler, such as looking for a different, hopefully bug-free version of the compiler (Heuristic 1), or debugging the compiler (Heuristic 3). Other solutions may include: reporting to the team leader, finding some kind of external technical support to resolve the problem (Heuristic 7), or using another language to finish the job if the junior programmer knows one (Heuristic 2). From the Articulator's point of view, these heuristics result in changes in the structure of `Model W`, such as replace resource allocation, adding an activity, or replace an existing activity. They also require dynamic scheduling since proper resource arrangement is needed for an SPM to be enacted efficiently.

Without considering circumstances, one might take any of above possible solutions to solve the problem. However, local constraints do limit choice of these solutions, as embedded in the selection heuristics. For example, if the model of articulation sets global selection heuristics No. 1 and 5 (Figure 6), which state that the junior programmer should use minimum extra tasks during articulation, the chosen problem-solving heuristic will be No. 7, even though the circumstance selection heuristics state that problem-solving heuristic No. 1 is a better choice.

Once problem-solving heuristic No. 7 is selected, the model of articulation uses it to create two solutions: reporting to the team leader and finding technical support. Although finding technical support might be more effective and time-saving, the model of articulation chooses to report the bug in the problematic compiler. This is because the global selection heuristics restricts extra work for the junior programmer and the other solution implies more work.

The final chosen solution from the model of articulation is to report to the team leader the discovery of bugs in the compiler. This solution transfers the problem to other agents without actually solving it. However, from the point of view of the junior programmer, it is no longer her problem anymore, and at the same time the solution satisfies her articulation heuristics. The Articulator implements the solution by adding a `report` activity before the broken activity and suspending the broken activity until the `report` activity gets a response from the team leader.

As time goes on, it might be possible that the compiler bug problem remains for a long period without being solved. The junior programmer becomes concerned because further delay may jeopardize her finishing the job, and eventually the whole project. Under the new circumstance, the model of articulation iterates and decides to use another solution, i.e. to find technical support by the junior programmer herself. In this case, the Articulator once again re-constructs `Model W` so that another `call-service` activity is added before the broken activity while abandoning the `report` activity. In this example, three different versions of this part of `Model W` have been created and recorded. They reflect articulation work under changed constraints at different times.

5 Conclusion

Software development can be better supported with the combination of software process modeling techniques and the model of articulation work. Planned activities are carried out under the guidance of SPMs with data and tool integration. In case a breakdown of planned activities occurs, the model of articulation work helps identify applicable solutions to recover or reorganize planned activities. The model of articulation work, at the same time, helps SPMs evolve during process enactment. As developers' understanding and experience about software processes grow, their applied SPMs become more realistic and reflect more local arrangements of software development. During this period, the model of articulation work serves as a means to accommodate changes within a project, or an organization into enhanced SPMs.

In sum, such a combination of software process modeling techniques and the model of articula-

tion work leads to a better foundation in process improvement and evolution.[5]

References

[BR88] V.R. Basili and H.D. Rombach. The TAME Project: Towards Improvement-Oriented Software Environments. *IEEE Trans on Software Engineering*, 14(6), Jun 1988.

[BS87] S. Bendifallah and W. Scacchi. Understanding Software Maintenance Work. *IEEE Trans on Software Engineering*, 13(3):311–323, Mar 1987.

[BS89] S. Bendifallah and W. Scacchi. Work Structures and Shifts: An Empirical Analysis of Software Specification Teamwork. In *Proc. of 11th International Conference on Software Engineering*, pages 260–270, Pittsburgh, PA, May 1989.

[CKI88] B. Curtis, H. Krasner, and N. Iscoe. A Field Study of the Software Design Process for Large Systems. *Communications of ACM*, 31(11):1268–1287, Nov 1988.

[CKSI87] B. Curtis, H. Krasner, V. Shen, and N. Iscoe. On Building Software Process Model Under the Lamppost. In *Proc. of 9th International Conference on Software Engineering*, pages 96–103, Monterey, CA, Apr 1987.

[Dow90] M. Dowson. Software Process Themes and Issues. In *Software Process Symposium*, Washington, DC, Sept 1990.

[FHF91] S. Fickas, R. Helm, and M. Feather. When Things Go Wrong: Predicting Failure in Multi-agent Systems. In *The AAAI Spring Symposium on Composite System Design*, Mar 1991.

[Gas86] L. Gasser. The Integration of Computing and Routine Work. *ACM Trans on Office Information Systems*, 4(3):205–225, Jul 1986.

[Gas91] L. Gasser. Social Conceptions of Knowledge and Action: DAI Foundations and Open Systems Semantics. *Artificial Intelligence*, 47(1-3):107–138, Jan 1991.

[GKC87] R. Guindon, H. Krasner, and B. Curtis. Breakdowns and Processes during the Early Activities of Software Design by Professionals. In G.M Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers (Second Workshop)*, pages 65–82. Ablex Publishing Corporation, 1987.

[GS86] E.M. Gerson and S.L. Star. Analyzing Due Process in the Workplace. *ACM Trans on Office Information Systems*, 4(3):257–270, Jul 1986.

[Hew91] C. Hewitt. Open Information Systems Semantics. *Artificial Intelligence*, 47(1-3):47–106, Jan 1991.

[HK89] W.S. Humphrey and M.I. Kellner. Software Process Modeling: Principles of Entity Process Models. In *Proc. of 11th International Conference on Software Engineering*, pages 331–342, Pittsburgh, PA, May 1989.

[5] Most recently, concepts and computational mechanisms for detecting and resolving related forms of articulation work in other domains are now under study by researchers in the area of distributed artificial intelligence [Hew91, Gas91, FHF91].

[HL88] K.E. Huff and V.R. Lesser. A Plan-Based Intelligent Assistant That Supports the Process of Programming. *ACM SIGSOFT Software Engineering Notes*, 13:97–106, Nov 1988.

[Kai88] G.E. Kaiser. Rule-Based Modeling of the Software Development Process. In *Proc. of the 4th International Software Process Workshop*, pages 84–86, New York, NY, 1988.

[KCI87] H. Krasner, B. Curtis, and N. Iscoe. Communication Breakdowns and Boundary Spanning Activities on Large Programming Project. In G.M Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers (Second Workshop)*, pages 47–64. Ablex Publishing Corporation, 1987.

[KS90] A. Karrer and W. Scacchi. Requirements for An Extensive, Object-oriented Tree/Graph Editor. In *Proc. ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 84–91. ACM Press, Oct 1990.

[MG90] N.H. Madhavji and V. Gruhn. Prism = Methodology + Process-oriented Environment. In *Proc. of 12th International Conference on Software Engineering*, 1990.

[MS90] P. Mi and W. Scacchi. A Knowledge-based Environment for Modeling and Simulating Software Engineering Processes. *IEEE Trans on Knowledge and Data Engineering*, 2(3):283–294, Sep 1990.

[MS91a] P. Mi and W. Scacchi. Process Integration in CASE Environments. Technical report, Computer Science Dept., USC, Los Angeles, CA, Jun 1991. Submitted for Publication.

[MS91b] P. Mi and W. Scacchi. Semantics of Process Enactment and Its Prototype Implementations. Technical report, Computer Science Dept., USC, Los Angeles, CA, Mar 1991.

[Ost87] L. Osterweil. Software Processes are Software Too. In *Proc. of 9th International Conference on Software Engineering*, pages 2–13, Monterey, CA, Apr 1987.

[Sca87] W. Scacchi. Models of Software Evolution: Life Cycle and Process. Technical Report SEI-CM-10, Software Engineering Institute, Carnegie-Mellon Unversity, Pittsburgh, PA, 1987.

[Str88] A. Strauss. The Articulation of Project Work: An Organizational Process. *The Sociological Quarterly*, 29(2):163–178, Apr 1988.

[Suc83] L.A. Suchman. Office Procedure as Practical Action: Models of Work and System Design. *ACM Trans on Office Information Systems*, 1(4):320–328, Oct 1983.

[TBCO89] R.N. Taylor, F.C. Belz, L.A. Clarke, and L. Osterweil. Foundations for the Arcadia Environment Architecture. *ACM SIGPLAN Notice*, pages 1–13, Feb 1989.

[War89] B. Warboys. The IPSE 2.5 Project: Process Modeling as the Basis for a Support Environment. Technical report, University of Manchester, Sep 1989.

[WEKC87] D.B. Walz, J.J. Elam, H. Krasner, and B. Curtis. A Methodology for Studying Software Design Teams: An Investigation of Conflict Behaviors in the Requirements Definition Phase. In G.M Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers (Second Workshop)*, pages 83–99. Ablex Publishing Corporation, 1987.