Articulation: An Integrated Approach to the Diagnosis, Replanning, and Rescheduling of Software Process Failures

Peiwei Mi and Walt Scacchi
Information and Operations Management Department
University of Southern California*
Los Angeles, CA 90089-1421
{pmi,scacchi}@gilligan.usc.edu

Abstract

The papers presents an integrated approach to articulate *software process plans that fail. Articulation repairs a plan when a diagnosed failure occurs and reschedules changes that ensure the plan's continuation. In implementing articulation, we combine diagnosis, replanning, and rescheduling into a powerful mechanism supporting adaptive process-based software development. Use of articulation in plan execution supports recovery and repair of unanticipated failures, as well as revising and improving the plans to become more effective. In this paper, we also describe how a prototype knowledge-based system we developed implements the articulation approach.*

1 Introduction

Software process plans specify the resources needed for the enactment of a software process model. They also specify the relationships between the resources to process steps, the products produced by these steps, and any resource or enactment constraints. Process plans guide the instantiation and use of process models by users or automated mechanisms. The use of knowledge-based plans and planning mechanisms for this domain was first introduced by Huff [HL88].

Plans that direct software development with limited resources can fail due to unexpected events. Planning mechanisms that must handle plan failure face two closely related problems: repairing the failed plan and recovering the broken plan execution. While the former is a dynamic planning or replanning problem, the latter involves dynamic or reactive scheduling. Most advanced planning and scheduling mechanisms tackle these two problems separately. For example, CHEF [Ham90] is a plan repair mechanism that explains and repairs failed plans in the domain of Szechwan cooking. SIPE [Wil88] also has a replanner to do the same thing in general problem-solving. However, they do not deal with rescheduling resources to meet changed arrangements. Dynamic schedulers such as [SDS90], on the other hand, primarily schedule resources incrementally in order to avoid unexpected events and backtracking. However, they are unable to identify arrangements that have been changed.

In this paper, we present an integrated approach to the diagnosis, replanning, and rescheduling of failed software process plans called *articulation*[1]. When a process plan fails during execution, articulation identifies problems based on knowledge of different types of plan failures. Articulation retrieves failure repair mechanisms from a knowledge base of problem-solving heuristics and implements them. Then articulation reschedules the necessary resources so overall plan execution is able to continue according to the modified plan. Our articulation approach is prototyped in a multi-agent knowledge-based system, called the Articulator [MS90, Mi92].

In what follows, we first give an overview of articulation in Section 2 which integrates diagnosis, replanning, and rescheduling. In Section 3, we briefly discuss plan representation in the Articulator. We present these issues and specify the problem space, the solution space, and the rescheduling mechanism in Section 4. In Section 5, we conclude the paper with a short discussion and summarize the unique characteristics

*Acknowledgements: This work has been supported in part by contracts and grants from AT&T, Hewlett-Packard, and Northrop Inc. No endorsement implied.

[1] Articulation originally refers to a kind of development work that people perform in order to continue their productive activities after a process breakdown occurs. It was first identified in a number of empirical studies of software development [CKI88, KS82, Str88]. The results of these studies are summarized and mapped into articulation heuristics elsewhere [Mi92].

found in the Articulator.

2 An Overview of Articulation

An overall view of the articulation approach that integrates diagnosis, replanning, and rescheduling is shown in Figure 1.

Plan execution starts when a specific process plan is instantiated. After resource scheduling is complete, *execution* starts to symbolically execute the activities step by step. Normally, it continues according to the execution order specified in the plan until all the activities are finished. However, the ongoing execution of a plan is blocked once a failure occurs. In the Articulator environment, articulation mechanisms now take over. The inputs for articulation are an executed plan with allocated resources, a broken activity in the plan, and an identified failure. The process of articulation consists of several stages: diagnosis, selection of problem-solving methods, recovery, and rescheduling.

In diagnosis, the Articulator identifies possible causes for a failure based on its symptoms and the other related information. The *problem space* in the Articulator stores domain-specific knowledge of failures that helps during diagnosis. From this point of view, the purpose of diagnosis is to locate a given failure in the problem space according to the situation where the failure occurs.

Once a diagnosis is identified, a solution must be created to resolve it. This stage is accomplished in the Articulator through selection of a pre-defined set of problem-solving heuristics (PSHs). However, all PSHs are not applicable at a given time, due to other constraints on resource availability. There are also other preferences that limit use of a particular PSH. All such issues of applicability and preference are abstracted into a set of *selection heuristics*. Selection heuristics are therefore applied during PSH selection to determine which PSH best matches the identified diagnosis and the current process enactment context.

The next stage is to apply the chosen PSH to *recover* from the failure. During this stage, the PSH is executed with a set of parameters, which define the current enactment context, gathered from diagnosis. Changes are then made in the broken activity, the plan, and/or the related resource allocation. At this point, the failure is resolved as far as the executed plan is concerned. However, this modified plan can not be executed directly because the original resource allocation as scheduled may not be valid due the changes. *Rescheduling* should then be performed to re-arrange the timing of the modified resource allocation. Otherwise, such changes may cause other failures.

Two outputs resulting from articulation are produced by the Articulator. First, a modified plan instance with allocated resources is put into execution. Second, a *repair suggestion* is forwarded to a user whose responsibility is to study sets of repair suggestions gathered from repeated plan executions in order to modify and improve to the original process model.

The Articulator is implemented in the form of a knowledge-based system[MS90]. The knowledge and experience of articulation abstracted from empirical studies of human articulation behavior are represented as knowledge schemata and inference rules[Mi92]. The Articulator has 4 modules that together perform support articulation:

- The *problem space* that embodies knowledge of failure and a *diagnosis mechanism* based on the problem space to interpret failures;
- The *solution space* that embodies knowledge of problem-solving methods in form of PSHs;
- A set of *selection heuristics* that bridges diagnosis to PSHs for given circumstances and application preferences; [2]
- A *rescheduling mechanism* that dynamically creates a global or partial resource allocation for a specified duration in response to changes in plans and related resources;

3 Plan Representation and An Example Plan

Before we get into the details of articulation, we briefly present our representation of software process plans and resource allocation. Also, we provide an example plan which will be referenced frequently in later sections.

In the Articulator, a software process plan is represented as a combination of several objects:

An *activity hierarchy* represents the decomposition of a plan into a hierarchy of smaller process steps called tasks and activities. Multiple levels of decomposition can occur depending on the complexity of a plan. At the bottom of this hierarchy are *activities*, which represent the smallest observable steps. We use

[2]Due to space limitations, we will not detail the selection process except to point out it is determined by applicability of PSHs to diagnosis as well as users preferences on use of defined PSHs. The details can be found in [Mi92].

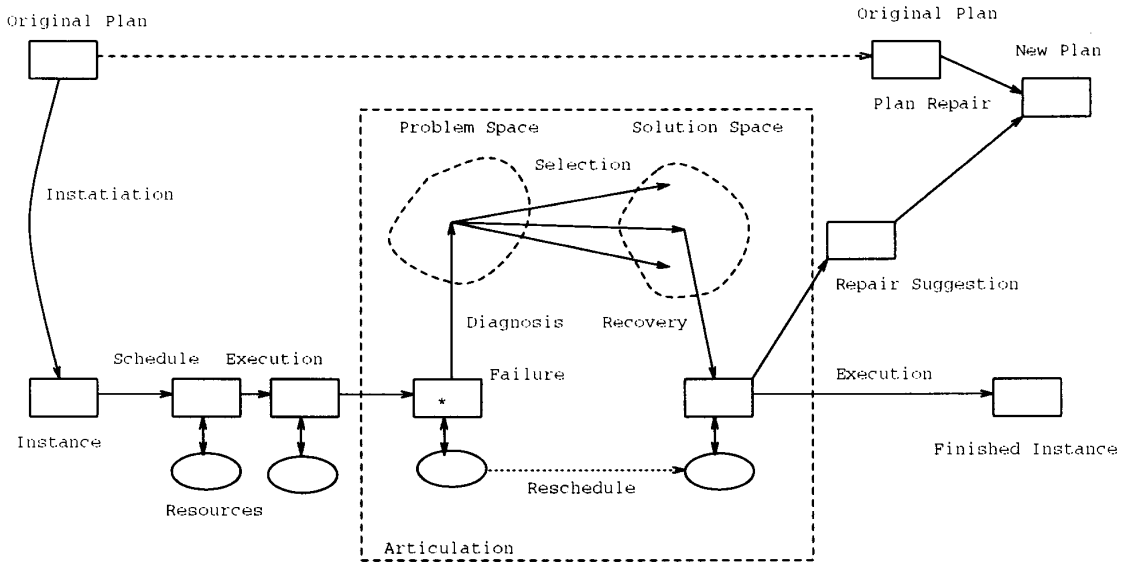


Figure 1: An Overview of Articulation

subtasks to refer to both tasks and activities since they are components in a plan. Within a level of a decomposition there exists a pair of relations to specify precedence order. These relations can define several types of execution ordering, such as linear, parallel, iterative, or conditional choice.

Resource requirements specify descriptions of the resources necessary to perform a subtask and produce its expected outputs. These requirements specify what resources, or types of resources, are needed in a subtask, but do not refer to particular instances of resources. Resource requirements for a subtask include: (i) information about the timing of subtask execution and expected duration; (ii) knowledge and skills required by the agents (or roles) performing the subtask; (iii) specification of tools and raw materials; and (iv) products that are created or enhanced during the subtask.

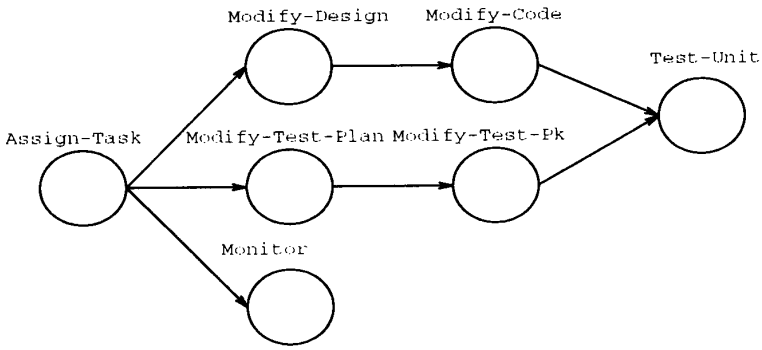
Resource possession includes "real" resource instances that are physically allocated and used during plan execution. It binds real resources to execution, while resource requirements do not. In order to have such a match of resource requirements and possession, each of the resource instances in resource possession should correspond to a particular entry in the resource requirement. In addition, these two resource relationships should be functionally equivalent in terms of their knowledge and skill requirements. As such, with these objects in mind, we provide an example.

Develop-Change-and-Test-Unit (DCTU) is an example of a software development process plan. This process modifies a software system's design and source code as part of a software development project[3]. It focuses on designing, coding, unit testing, and management of a localized change to a software system.

Figure 2 gives a graphic display of the activity hierarchy of the **DCTU** process as well as the definition of activity **Modify-Design**. **DCTU** is decomposed into a set of 7 activities shown as a directed graph in the figure. For example, once **Assign-Task** is done, three activities, **Monitor-Progress**, **Modify-Test-Plan**, and **Modify-Design** can be started.

Resource requirements are specified at the level of activities. In Figure 2, resource requirements for activity **Modify-Design** is given. It states that **Modify-Design** requires two kinds of agents: **Design-engineer** and **Software-engineer** as well as some other kinds of required-resources. In defining these requirements, we assume that object classes, such as **Design-engineer**, have been previously defined in the knowledge base.

[3]This example was created by the organizing committee of the 6th International Software Process Workshop in 1990[KFFe90]. The original problem served as a standard modeling problem for comparing various existing approaches to software process modeling. We made minor changes to the original problem and reuse it here as an example of articulation.



{{Modify-Design	instance:	activity
	task-component-of:	Develop-Change-and-Test-Unit
	task-has-predecessor:	Assign-Task
	task-has-successor:	Modify-Code
	task-has-agent-role-spec:	agent-for-modify-design
	task-has-required-resource-spec:	required-for-modify-design
	task-has-provided-resource-spec:	provided-for-modify-design}}
{{agent-for-modify-design	instance:	agent-role-spec
	agent-role-spec-in-task:	Modify-Design
	resource-requirement:	Design-engineer Software-engineer
	maximum-quantity:	1 1
	resource-possession:	John Doug}}
{{required-for-modify-design	instance:	required-resource-spec
	required-resource-spec-in-task:	Modify-Design
	resource-requirement:	System-Design-doc Requirements-change
	maximum-quantity:	1 1
	resource-possession:	System-Design-doc Requirements-change}}

Figure 2: Partial definition for **Develop-Change-and-Test-Unit (DCTU)**

Let us assume that a team of software engineers and other related resources are allocated for **DCTU**. For **Modify-Design**, John is assigned as a **Design-engineer** and Doug as a **Software-engineer**. For other classes of resources, we use the same name to represent instances within the classes for this example.

Although it is an oversimplified example, it is easy to anticipate that given different sets of resource possession, plan execution will be different. It is also easy to see that whenever resource possession changes, plan execution changes accordingly. These relations give rise to the necessity for articulation, and therefore, the role of the Articulator is to handle unexpected changes or failures during process plan execution.

4 Diagnosis, Recovery, and Rescheduling

Diagnosis, recovery, and rescheduling are three critical activities in articulation. In this section, we briefly discuss these activities along with the example presented previously.

Once a failure occurs, diagnosis locates a position of the failure in the problem space (Figure 3), which consists of three dimensions: Failure type, Usage of resource, and Task type. At the same time, a necessary explanation is gathered to support the classification. Diagnosis then searches for a causal explanation that matches the problem space. The position of a failure will then be used as a pointer to the solution space, which identifies a set of possible PSHs.

Diagnosis identifies the type of failure by answering following questions through both direct information retrieval and deductive reasoning, as provided by the Articulator[MS90]:

Failure Type	*Usage of Resource*	*Task Type*
Resource not created	Activity	Intra
Unnecessary resource	Agent	Inter
Unspecified resource	Tool	
Unallocated resource	Required-Resource	
Inadequate resource match	Provided-Resource	
Occupied resource		
Broken resource		

Figure 3: The Problem Space

- Is the plan execution complete?
- Is there a problematic resource associated with a failure?
- Is there a problematic resource requirement?
- Is there a match between the problematic resource and its requirement?
- What resources does the broken activity provide?

We will not go into a detailed description of the implementation of diagnosis, but simply point out that it is similar to those suggested in [Ham90]. In total, there are 25 diagnosis strategies with their corresponding values in the problem space [Mi92].

Let us now consider an example failure as follows: As we see in DCTU (Figure 2), **Modify-Design** is immediately followed by **Modify-Code**. When execution starts **Modify-Design** begins with two assigned agents, John and Doug. **Modify-Design** creates a new version of **System-Design-Doc** that is suppose to satisfy the changes in system design. The execution then initiates **Modify-Code** which is assigned to Doug and Chris. While modifying the source code, Doug and Chris may find that the modified design document has flaws that create new problems for the software system, subsequently making **System-Design-Doc** inappropriate to follow when modifying the system's source code. Therefore, they identify an **inadequate-match** and hand it over the Articulator.

Once the failure is located in the problem space, the Articulator acts to resolve the problem. The solution space in the Articulator is an abstraction of methods that recover failures. The solution space contains various *problem-solving heuristics (PSHs)* that fix or avoid failures. These PSHs are characterized by four dimensions that identify their operations and operand objects (Figure 4): type of operation, target resource usage, operand usage, and operand. When a PSH is selected, it is important to know both the type of the operation and the operand objects for the operation. The four dimensions in the solution space are designed to make these characteristics explicit.

In sum, a template of all four dimension values can be used to index a particular PSH, which is used to link to a particular diagnosis. Consider the following example:

```
<replace-instance, required-resource,
 required-resource, existing-other>
```

and

```
<redo-and-review, required-resource,
 activity, new>
```

This example identifies two PSHs that can be used to repair the example diagnosis given in the previous section. The first PSH searches for another copy of **System-Design-Doc** to replace the problematic one. The second PSH adds more activities to modify the problematic **System-Design-Doc**.

Without going into the details of PSH selection, we simply point out that the selection process uses global and local constraints to identify an applicable PSH among all defined PSHs. The applicable PSH is then executed to fix a given diagnosis. In this example, the second PSH is chosen to be applicable because it is a more feasible solution for the given circumstance.

The Articulator's PSHs owe a great deal to previous work on plan repair ([Ham90, Wil88]). However, PSHs here are oriented to practical methods that are gathered from the empirical studies[Mi92], rather than a collection of single replanning actions. To this end, a PSH contains a set of changes that prove effective to a particular problem. Also, PSHs are organized such that a single failure can have multiple PSHs applied to it. As an example, consider the chosen PSH

```
<redo-and-review, required-resource,
 activity, new>
```

It represents a combination of replanning actions that creates a pair of activities and inserts them into the broken plan. Figure 5 gives an algorithmic description of this PSH. Other defined PSHs are implemented in similar fashion as inference rules.

Type of Operation	*Target Usage*	*Operand Usage*	*Operand*
Replace-Instance	activity	activity	existing-self
Replace-Class	agent	agent	existing-other
Restructure	tool	tool	new
Modify	required-resource	required-resource	
Redo-and-Review	provided-resource	provided-resource	
Split/Merge			
Others			

Figure 4: The Solution Space

```
1) Create another instance of the activity to be redone;
2) Create a new instance of activity Review and add resource requirements;
3) Link the two as activity 1) proceeds activity 2);
4) Insert 3) into the plan just before the broken activity.
```

Figure 5: Algorithmic Description of **Redo-and-Review**

When this PSH is executed on our example, a modified plan is created as in Figure 6. When the PSH is executed, the first half of the plan has been completed and the second half is in progress. Furthermore, this modified plan is not yet ready for execution since its resource allocation is not yet done for the new part. As such, we turn to explain the rescheduling process that completes the remaining part before execution re-starts.

Articulation requires scheduling to be reactive and partial. First, scheduling is called in response to changed resource arrangements. The previous arrangement must be abandoned and new arrangements asserted. Second, only part of the whole schedule needs to be modified while others remain the same. As such, changes should be limited as much as possible. In the light of these two requirements, a rescheduling mechanism has been implemented in the Articulator that is interfaced to the other mechanisms. The Articulator's rescheduling mechanism is based on heuristic constraint-directed search[FSe89]. It has a similar structure as those just cited, but it is designed to be reactive and partial. The rescheduling mechanism implements two types of constraints for filtering tasks: *unary rescheduling constraints* and *binary rescheduling constraints*, as well as two types of heuristics for reducing search space and backtracking: *search heuristics* and *resource heuristics*[Mi92].

The rescheduling mechanism takes over once a PSH is executed. First, the rescheduling mechanism decides which part of the previous schedule to abandon and which resources to reschedule. This part is very important since it acts as a link between replanning and rescheduling. It is to our advantage that this is done with reference to the executed PSH and changes made during PSH execution. When deciding the part to abandon, there are several alternatives. One strategy is to discard only those resource possessions for the broken activity and all its successors. This strategy is intended not to disturb part of the old schedule that binds the resources to other unaffected activities, and therefore limits the scope of changes. Another possible strategy is just to abandon all resource allocations that have not been used and schedule all remaining activities altogether. This second strategy creates more potential changes to the original plan, but may have a better result.

Now, let us consider our example of DCTU for rescheduling (see Figure 6 for its activity hierarchy). Part (a) in Figure 7 is the original agent schedule. In this schedule, we assume that most activities take 1 time unit to complete while **Monitor-Progress** and **Modify-Test-Plan** take longer. We also have an agent-to-role binding according to their skills. Part (b) in Figure 7 reflects a modified schedule created for **Modify-Design** and its successor activities. In this new schedule, we assume that **Review-Design** is assigned to four agents, Peter, John, Doug, and Chris. Their presence is required for the activity to start. Part (c) in Figure 7 is created by first discarding part of the schedule for all unstarted activities at time 2 and scheduling them together with the newly added **Modify-Design** and **Review-Design** activities. Accordingly, this shows the final schedule (c) takes fewer time steps to complete than (b) because of a better task arrangement for Peter. Finally, at this time, the rescheduling mechanism does not optimize modified schedules, but it is equipped with the capability that allows users to try out different strategies.

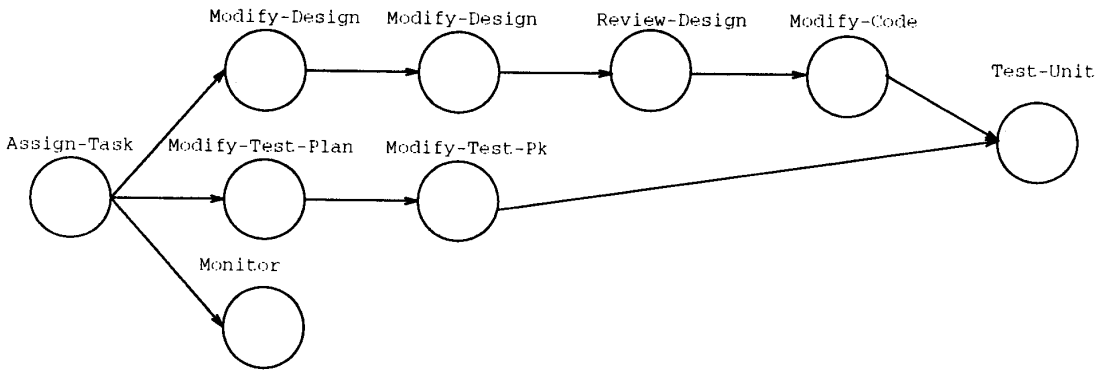


Figure 6: The Modified **Develop-Change-and-Test-Unit**

Time	Mary	Peter	John	Doug	Chris
	Project-Manager	QA-Engineer	Design-Engineer	Software-Engineer	Software-Engineer
(a) *The Initial Schedule*					
1	Assign-Task				
2	Monitor-Progress	Modify-Test-Plan	Modify-Design	Modify-Design	
3	Monitor-Progress	Modify-Test-Plan		Modify-Code	Modify-Code
4	Monitor-Progress	Modify-Test-Package			
5	Monitor-Progress	Unit-Test	Unit-Test		
(b) *The Modified Schedule (I)*					
1	Assign-Task				
2	Monitor-Progress	Modify-Test-Plan	Modify-Design	Modify-Design	
3	Monitor-Progress	Modify-Test-Plan	Modify-Design	Modify-Design	
4	Monitor-Progress	Modify-Test-Package			
5	Monitor-Progress	Review-Design	Review-Design	Review-Design	Review-Design
6	Monitor-Progress			Modify-Code	Modify-Code
7	Monitor-Progress	Unit-Test	Unit-Test		
(c) *The Modified Schedule (II)*					
1	Assign-Task				
2	Monitor-Progress	Modify-Test-Plan	Modify-Design	Modify-Design	
3	Monitor-Progress	Modify-Test-Plan	Modify-Design	Modify-Design	
4	Monitor-Progress	Review-Design	Review-Design	Review-Design	Review-Design
5	Monitor-Progress	Modify-Test-Package		Modify-Code	Modify-Code
6	Monitor-Progress	Unit-Test	Unit-Test		

Figure 7: Schedule for **Develop-Change-and-Test-Unit**

5 Discussion and Conclusion

Before we conclude this paper, let us examine our example to see what kind of repair the discussed failure brought. Initially, there is no iteration defined in the plan (Figure 2). After a failure of **Inadequate-match** is identified, the Articulator introduces a repeated **Modify-Design** and a new **Review-Design** to the plan (Figure 6). Subsequently, the Articulator suggests an iteration to encapsulate the two activities in the generated repair suggestion. If a manager accepts this suggestion and incorporates it into the original plan, this new plan takes care of a problem that was not originally addressed. Therefore, similar failures could be avoided in future instantiations of this process.

Currently, the articulation approach is applied in both modeling and designing software development processes[MS90, Mi92]. On the one hand, it supports simulation of software process plans as a means to debug and tailor them. On the other hand, it enables developers to recover from process breakdowns during software development which were not anticipated in software process plans. To this end, we have built two separate systems for these two purposes respectively. The Articulator system provides the capability for handling the dynamic evolution of software engineering processes. The second system, our process-driven CASE environment, accepts a software process plan from the Articulator as its input, displays it in a graphic form to its assigned developers, and sup-

ports data management, tool integration, and project management during software development[MS92]. As such, when users in their assigned agent roles experience a process plan breakdown while using this CASE environment, they can then forward the identified failure back to the Articulator for resolution. After articulation is completed, the Articulator then outputs a revised process plan which is then input to this CASE environment.

In conclusion, articulation of failed plans in execution is an integrated mechanism for diagnosing, replanning, and rescheduling software process plans that fail during enactment. Our approach was influenced by works on both planning and scheduling [Ste81, FSe89]. Further, CHEF [Ham90] and SIPE [Wil88] provided us with insight on the depth of plan repair and some valuable replanning actions. However, compared with these approaches, the Articulator and its articulation mechanisms are unique in three aspects: First, it is designed to handle both plan repair and rescheduling. This results in an integrated system of replanning and rescheduling which is powerful and convenient to users. Second, the Articulator relies heavily upon knowledge and skills for repairing problems in the domain of software development. The kind of knowledge and skills is abstracted from a number of empirical studies by us and others. It is also implemented as an open system so that more heuristics can be added easily. Finally, our articulation mechanisms were conceived to help solve process failure problems, which will likely become more prevalent as both conventional and knowledge-based software engineering environments evolve to support process-centered methods for software development. As such, our strategy for organizing articulation problem and solution spaces, together with problem solving hueristics and PSH selection hueristics, provides a viable and extensible foundation for further exploring how to better support knowledge-based software development.

References

[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.

[FSe89] M.S. Fox, N. Sadeh, and etc. Constrained Heuristic Search. In *Proc. of Joint International Conference on Artificial Intelligence*, pages 309–315, 1989.

[Ham90] K.J. Hammond. Explaining and Repairing Plans that Fail. *Artificial Intelligence*, 45(3):173–228, 1990.

[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.

[KFFe90] M. Kellner, P.H. Feiler, A. Finkelstein, and etc. Software Process Modeling Example Problem. In *The 6th International Software Process Workshop*. Japan, Oct 1990.

[KS82] R. Kling and W. Scacchi. The Web of Computing: Computer Technology as Social Organization. In *Advances in Computers, Vol.21*, pages 1–90. Academic Press, Inc., 1982.

[Mi92] P. Mi. *Modeling and Analyzing the Software Process and Process Breakdowns*. PhD thesis, Computer Science Dept. University of Southern California, 1992. September.

[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, Sept 1990.

[MS92] P. Mi and W. Scacchi. Process Integration in CASE Environments. *IEEE Software*, 9(2):45–53, March 1992.

[SDS90] M.J. Shah, R. Damian, and J. Silverman. Knowledge Based Dynamic Scheduling in a Steel Plant. *IEEE Sixth Conference on Artificial Intelligence Applications*, pages 108–113, 1990.

[Ste81] M. Stefik. MOLGEN Part 1: Planning with Constraints. *Artificial Intelligence*, 16(2):111–139, May 1981.

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

[Wil88] D.E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann Publishers, Inc., 1988.