A Knowledge-Based Environment for Modeling and Simulating Software Engineering Processes

PEIWEI MI AND WALT SCACCHI, MEMBER, IEEE

***Abstract*—We describe the design and representation schemes used in constructing a prototype computational environment for modeling and simulating multiagent software engineering processes. We refer to this environment as the Articulator. We provide an overview of the Articulator's architecture which identifies five principal components. Three of these components, the knowledge meta-model, the software process behavior simulator, and a knowledge base querying mechanism, are detailed and examples are included. The conclusion reiterates what is novel to this approach in applying knowledge engineering techniques to the problems of understanding the statics and dynamics of complex software engineering processes.**

***Index Terms*—Agents, artificial intelligence, deductive query, distributed probem solving, meta-model of software processes, modeling of software processes, process programming, process simulation.**

I. Introduction

MODELING the process of software engineering represents a promising approach toward understanding and supporting the development of large-scale software systems. The *software process* is the *collection of related activities, seen as a coherent process subject to reasoning, involved in the production of a software system* [26]. A *software process model* is a prescriptive representation of software development activities in terms of their order of execution and resource management. A *software process meta-model* is a representation formalism which provides necessary components to create various types of software process models [26].

A meta-model of the software process should possess the capability to include major properties of contemporary software development practice. Recent evaluations on software process models [4], [20] suggest that effective software process models should address organizational and technical dimensions including 1) detailed descriptions of software processes, products and settings; 2) their interactions; 3) management and exception handling during the performance of software processes; and 4) product-specific, organization-specific, and project-specific processes. We present a meta-model which uses a knowledge representation language to specify all these aspects and further provides mechanisms to investigate the interactions among these dimensions.

An automated modeling environment for software development should be powerful enough to support model validation and verification. By simulating a specified software process, the environment and its users can collectively detect faults, inconsistencies, or anomalous behavior in a process prescription. Emerging conflicts in time schedule and resource allocation, for example, are some common anomalies in multiagent process plans [6]. Complex faults, on the other hand, may concern the configuration of task decomposition and organizational settings as well. The environment should also assist in determining possible solutions for contingencies encountered in task execution. These solutions are based on particular resource and knowledge configurations, hence they should be setting-specific, project-specific, agent-specific, and time-specific. As such, by simulating task execution, this enables a user to predict development progress on a more realistic basis and compare different process models. We describe the design of such an environment which utilizes our software process meta-model.

In the next section, we provide some background to our approach. In Section III, we present the system architecture of the Articulator and discuss important issues covered in the design and use of the Articulator. Following this, we will discuss some of the subsystems in turn: Section IV discusses the knowledge base, which stores our meta-model of software processes, Section V gives accounts for the simulation of the Articulator meta-model, and Section VI presents the query mechanism. We then conclude with a summary of novel contributions of the Articulator project.

II. Background

As we noted earlier, there has been growing interest focused on the problem of modeling, analyzing, and automating various aspects of software engineering processes [22]. Wileden [26] suggested a modeling framework based upon use of a software process meta-model. Osterweil followed with a paradigmatic approach which casts the software process meta-model into what he called a *process programming language*—a language for programming prescriptive process models into a software development environment [19].

Since then, much research effort has been directed to the design and implementation of languages for software

Manuscript received November 30, 1989. This work was supported in part by contracts and grants from AT&T, Bellcore, Pacific Bell, and the Office of Naval Technology through the Naval Ocean Systems Center. No endorsement implied.

The authors are with the Computer Science Department, University of Southern California, Los Angeles, CA 90089.

IEEE Log Number 9036986.

1041-4347/90/0900-0283$01.00 © 1990 IEEE

process automation, and to the construction of more realistic models. For example, many researchers have introduced process language constructs including rules and pattern matching [16], behavioral patterns [27], graphic finite-state machines [15], and agent-task-product relations [8]. But none provides a direct means for querying the status or state of a modeled software process. Others including [14] and [20] use knowledge representation languages and deductive planning mechanisms for software process modeling. But overall, these efforts lead to closed, single agent (i.e., globally controlled) systems. Furthermore, with the exception of [15] and [8], most efforts do not explicitly reference or use empirical sources for their software process models.

Modeling and simulating complex organizational processes performed by people requires an empirically based, multiagent open systems framework [11], [13]. For instance, [5] and [2] are examples of recent empirical studies aimed at providing more realistic descriptions of multiagent software processes. But their modeling efforts have not been cast in the form of a language or computational environment.

We seek to resolve these shortcomings in modeling and automating (i.e., simulating) software processes. This allows us to identify what is new about our work presented here. Our approach uses a software process meta-model derived from an established approach for empirical studies of computing work in organizational settings. In addition, it allows us to model multiagent software processes in an open systems manner, meaning that process conflicts can arise that must be resolved locally (i.e., through agent-agent interactions), rather than through automated global control. The environment supports the simulation of these multiagent process models. The meta-model, individual process models, and process simulation traces can each be queried both directly and deductively. In the sections that follow, we will describe each.

III. The Architecture and Users of the Articulator

The Articulator is a knowledge-based environment for studying software processes. It provides a meta-model of software processes, an object-based language to specify models of software processes, and an automated simulation mechanism. The system architecture of the Articulator consists of five subsystems (Fig. 1): the knowledge base, the behavioral simulator, the query mechanism, the instantiation manager, and the knowledge acquisition manager. The Articulator has been prototyped over a two-year period using the *KnowledgeCraft*™ (KC) knowledge engineering environment on a *TI-ExplorerII*™ [3].

The *knowledge base* implements the Articulator meta-model by an object-based approach. The meta-model consists of the web of resources and situations, which is a model of software development, the representation of agent's task performance skills. The Software Process Specification Language (SPSL) is a user interface enabling users to define customized software process models based on the meta-model. The knowledge base is defined within an object-based knowledge cluster. Section IV provides more details of the Articulator knowledge schema.

The *instantiation manager* manages the relationships between the meta-model, the customized software process models, and their instances. It maintains all these relationships according to creation time and lines of inheritance, as well as retrieving the correct instance when requested. In contrast to conventional database systems, there is no explicit boundary among the meta-model, the software process model, and their instances. All of them can be manipulated and modified as an associated instance to the old one whenever needed.

The *behavioral simulator* controls the simulation of a given software process model and creates a process trajectory[1] over a development period. In this regard, behavioral simulation is a symbolic execution of a software process model described in SPSL notation. Mechanisms to perform software process activities are implemented within the behavioral simulator. In terms of the representation of software processes, there are three types existing in the Articulator: the prescriptive process model before execution, the simulated trajectory of the prescription, and the descriptive recorded development history. Each of them serves a different purpose in the Articulator, but bears the same form of representation. Section V provides a more detailed view of the behavioral simulator.

The *query mechanism* supports logical rules for various types of deductive queries. It helps users access information in the Articulator efficiently. The information sources are the knowledge base, i.e., the meta-model, the software process models, and their instances. Using object inheritance techniques and backward-inference mechanisms, the query mechanism reasons about information and knowledge to determine answers to several types of questions. See Section VI for a description of the query mechanism.

The *knowledge acquisition manager* is an interface for the Articulator to get a software process model and associated data. The knowledge acquisition manager takes a structured description of agents, tasks, and resources as inputs, then translates it into a configured software process model. It also assesses the information gathered from software development projects controlled by the Articulator and stores them for later use. An automated knowledge acquisition manager for interactive capturing of model refinements and real-time software process data has not been implemented yet, but simple mechanisms for model and data input are currently in use.

Users of the Articulator fall into three categories: process researchers, project managers, and software developers. *Software process researchers* study software process models in order to identify ones that are highly efficient, satisfy different performance requirements, or reveal subtle software process dynamics requiring further

[1]A process trajectory is a sequence of snapshots of software development over a period of time.

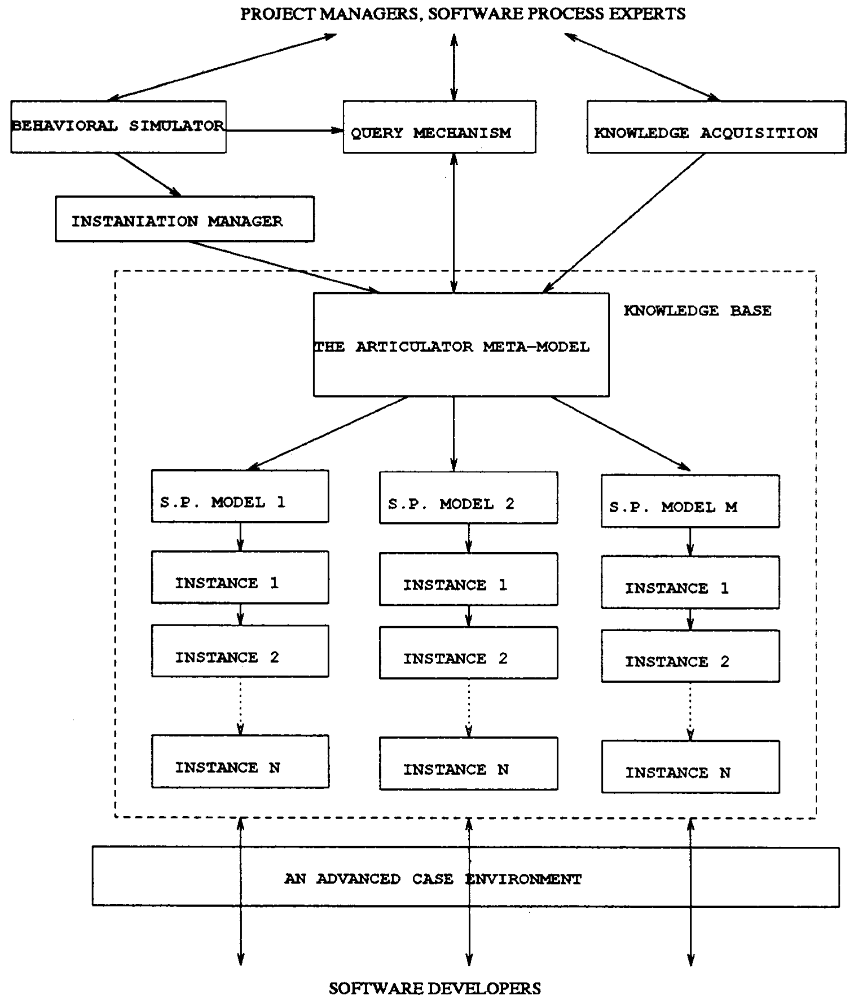


Fig. 1. The system architecture of the Articulator.

study. These users define various types of software process models, test them by simulation, compare the simulation results to observed development histories, and refine them according to certain criteria. This is an iterative, incremental process; it continues until an acceptable model of software processes is achieved which fits in a particular infrastructure. These users may potentially modify the meta-model as well when it is necessary to incorporate new features into it.

Software project managers select or configure an existing software process model which suits their project needs and essentially provides guidance (a plan) for how to carry it out successfully. Managers use the Articulator to get access to a software process model, instantiate it according to their local project situations, simulate and refine it in order to create a plan for development, and realize it in their own organizational settings. When they encounter unexpected problems during a planned development project, they can consult the Articulator to find plausible solutions and to evolve the model based on the solution. This is similar to the suggested use of the Callisto system used in computer manufacturing processes [20].

Software developers use the Articulator through a CASE environment [22]. In this way, the Articulator helps to coordinate their development activities according to a prescriptive model, and serves as both an active agenda mechanism and an information exchange center. At the same time, all the development activities are recorded into a history of development and can be fed back to managers in order to monitor development progress. This history will then be used by software process reseachers as an empirical source of observation on the practical character of the software process model in use. It can also be used in validating the model or in making modifications to it.

IV. The Articulator Meta-Model

This section presents the Articulator meta-model stored in the knowledge base. It is an object-based representation of a software development infrastructure that consists

of the web of resources and situations and the agent's task performance skills.

A. *The Web of Resources and Situations*

The web of resources and situations describes an infrastructure of developers, organizations, tasks, and other resources, through which software systems are engineered. It is intended to provide an articulate view of the many aspects of software engineering processes within a single formalism. The theoretical scheme underlying the Articulator meta-model is the web model of computing introduced by Kling and Scacchi [17]. This web model replies upon empirical studies to make explicit a variety of connections between computing technologies, artifacts, activities, together with their embedding social situations and organizational infrastructure. It also focuses equal attention to how people and their computing systems interact, cooperate, compete, and conflict with each other in the course of their work.

In computational form, the web consists of clusters of attributed objects and relations linking them, together with various processing mechanisms. Each of the objects is defined as a model of a type of the components in the software process and represented as a schematic class. These objects are further divided into several subclasses. A subclass is divided repeatedly until an empirically observable level of detail is reached. Furthermore, every schematic class has a set of attributes specifying its own properties, a set of relations linking to other classes, and may have many instances to inherit its defined properties and relations with their defined values. Several high level classes in the web of resources and situations are shown in Fig. 2. We briefly discuss its main components and their relationships here.[2]

The top level abstraction of the Articulator meta-model consists of three major objects: resources, agents, and tasks. They are linked together through two relations: agents *perform* tasks and tasks *use* resources. This abstraction captures our fundamental understanding of software development processes, and in a larger sense, complex organizational activities.

A *resource*, as a model of general objects and products, portrays the general properties of organizational objects and is the root of the Articulator meta-model in terms of the IS-A relation. Accordingly, resources are objects used in tasks by agents. In the Articulator meta-model, tasks consume and produce resources, which alter the values of resource attributes. These attributes include, among others, name, current status, functional description, location, ownership, and usage in tasks and by agents.

An *agent* represents a collection of behaviors and associated attributes. An agent's behavior emerges during the performance of tasks (including communications, accommodation, and negotiation) given the agent's set of skills, available resources, affiliated agents, and organizational constraints or incentives—that is, given the agent's circumstantial situation. We use agents as a general model of developers, development teams, and organizations.

An agent's ability to perform tasks is defined by its working load, its agenda, its selected work style, and its working tasks together with its behavior controller (its "self"). An agent may also have skill, experience, and knowledge of task performance. In order to perform a task, an agent must possess the necessary resources and rights of information access. Living in an organizational infrastructure, an agent may have affiliations with other organizations and play different roles in different organizational situations [8], [17]. Besides these, agents have a knowledge representation which specifies their potential behaviors, and this behavior can also be dynamically simulated. These aspects will be discussed later.

There are several types of agents in the Articulator meta-model. *Individual agents* are single entities, such as a single developer or a single machine. *Collective agents*, such as *teams* and *organizations*, have infrastructures defined for a group of agents to work together, thereby enlarging their efforts. In a collective agent, individual agents work cooperatively or competitively to achieve their collective and individual goals. However, collective agents can also be in conflict over how to achieve their goals, as well as over which goals are worth achieving in what order.

A *task* models organizational work and development processes. Tasks represent situations for work and processes in terms of a network of actions (i.e., operators) that agents perform which manipulate the web of resources and situations. A task, defined as a structural hierarchy, is used to represent both a semi-formal plan of the actual task before it is carried out (a prescription) and the actual execution trajectory of the task after it has been done (a description). Both of these include a hierarchy of task decomposition and a nonlinear performance sequence. We model two types of organizational work: primary tasks and articulation tasks [1], which distinguish development-oriented tasks from coordination-oriented tasks. The hierarchy of task decomposition may include multilevel nested decomposition, iteration, and multiple selection. Levels of specification depend on user requirements and can be modified as requested. At the bottom level of this hierarchy are actions. *Actions* are basic processing units within the processing mechanism. Furthermore, an action links to a procedural specification, such as a LISP function or a forward-chaining mechanism, which propagates updates through the current state (or instance) of the web of resources and situations.

Interesting properties of a task include: assigned and authorized performers; task hierarchy and execution ordering; schedule; duration, deadline, start time and finish time; and resources planned to be consumed or produced by the task.

[2]The current implementation of the web represents more than 500 object classes and nearly 2000 relations. Most object classes include ten or so attributes. In addition, there are over 200 rules and procedures which support behavioral simulation and query processing.

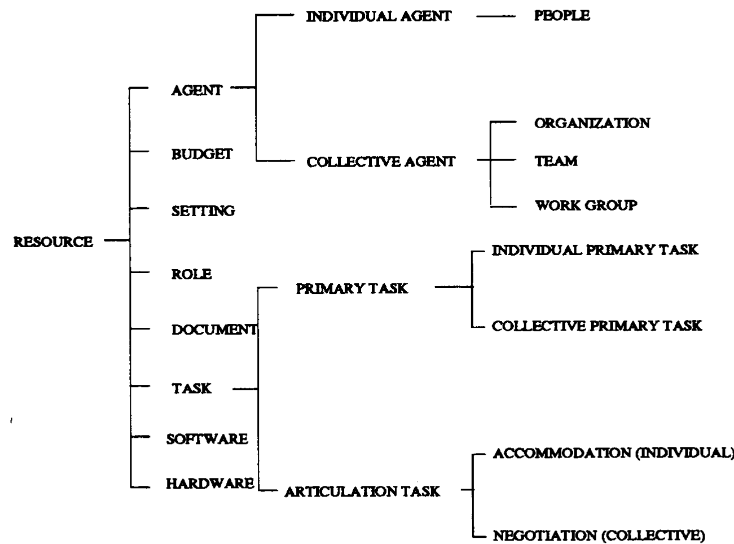


Fig. 2. Part of the web of resources and situations along the IS-A relation.

The Articulator meta-model is an open system [13]. It has the following special characteristics.

- The boundary of the meta-model and its interface with the outside world is determinable, though not necessarily static. The meta-model, besides manipulating its own resources, communicates with the outside world. Such communication of the Articulator meta-model with the outside world is made possible through acquiring or providing resources.
- All the resources in the web have their own life cycle. Every instance of a resource is either created by some task, or introduced by the outside world, persists for a period of time, and then is consumed by other tasks or exported to the outside.
- An agent's power to manipulate the web of resources and situations is limited and configurable. This manipulation power includes possession or control of resources, rights of information access, and rights of task performance. This power can be restricted to some constraint over a period of time. On the other hand, this power may be reconfigured at any time by authorized agents. In this way, centralized control, distributed control, or something in between, can be modeled in the Articulator meta-model. Also, differences in relative power among interacting agents can give rise to conflicts in task performance. These conflicts can thus alter the task situation from focus on performance to resolution of conflict.
- The web of resources and situations is a densely interrelated infrastructure. By definition, any entity in the web is associated with many other objects through relations. With this kind of infrastructure, execution of a task can cause many implicit side effects besides its intended behavior. For example, in a development task, a manager agent may assign a task to a developer agent without allocating the necessary resources for task completion. This will not cause any problem in task assignment, but will surely delay the task execution since the developer agent will have to spend time to find the resources required for task execution. The consequence and implication of side effects are of interest because they resemble real situations in many ways.

Establishing a model of a software development process is made possible by using the Articulator meta-model. Different types of software process models can be defined. For example, a software production-process model, such as the Waterfall model [8], [15], [19], [27] or the Automation model [8] can be specified by the Articulator as a hierarchy of software development activities and their suggested prescriptive execution sequence. A software production-setting model can be viewed as a mixed task representation of primary tasks and articulation tasks. As an example to be used later, we define a simple working team here. A development team, called Team-A, belongs to company F and has three members: Mary, Joe, and Peter. The team is responsible for the task of designing the FOO system, which consists of two component tasks: architecture design and detail design. This small example essentially shows a setting, a development team, and a task assigned to the team. Fig. 3 gives a specification of the example in SPSL, while other details, such as resource specification, detailed process prescription will be provided later.

B. *Model of Agent's Task Performance Skill*

An agent's behavior during the software process is the way it performs tasks, given its plans and emerging circumstances. In other words, it is the trajectory of task execution that constitutes the agent's behavior. The behavioral specification is a knowledge representation of task performance skill [24]. An agent's task performance skill is represented according to a three-level paradigm, where each level is a space specifying a particular type of

```
;; Company-F is an organization in the model.
(define-object Company-F
   (is-a ORGANIZATION))

;; Team-A is a team in Company-F and has two members, Mary and Joe.
(define-object Team-A
   (is-a TEAM)
   (team-belong-to-organization Company-F))

(define-object Mary
   (is-a PEOPLE)
   (individual-in-collective-agent Team-A)
   (task-execution-strategy Finish-one-FIFS)
   (accommodation-strategy Switching))

(define-object Joe
   (is-a PEOPLE)
   (individual-in-collective-agent Team-A)
   (task-execution-strategy Finish-one-FIFS)
   (accommodation-strategy Waiting))

;; Task Design-FOO has two subtasks and is assigned to Mary and Joe.
(define-object Design-FOO
   (is-a TASK-CHAIN)
   (task-force-assigned-to-agent Joe Mary)
   (production-task-has-component Architecture-design Detail-design))

;; Subtask Architecture-design is assigned to Mary
(define-object Architecture-design
   (is-a TASK-CHAIN)
   (task-force-assigned-to-agent Mary))

;; Subtask Detail-design is assigned to Mary and Joe
(define-object Detail-design
   (is-a TASK-CHAIN)
   (task-force-assigned-to-agent Mary Joe)
   (task-force-has-predecessors Architecture-design))
```

LEGEND:
Lower-case words: Reserved key terms.
UPPER-CASE words: Reserved object types.
Upper-case starting word: Defined objects.

Fig. 3. An SPSL specification of Team-A. Five classes of objects are defined in terms of their associated attributes. Other details are omitted for simplicity. The task specification performed by this team is given in Fig. 7.

knowledge and operators to manipulate it. This three-level paradigm is similar in concept with other multilevel problem-solving architectures, such as those in [10] and [25].

The *domain space* stores information and knowledge of an application domain. This is an agent's *personalized* domain knowledge and is generally a subset image of the web. It is limited by the agent's manipulation power, i.e., its possession of resources, its information access, and its rights of task performance. Operators in the domain space are actions that a designated agent can perform within the application domain.

The *task space* stores operational knowledge of the manipulation and reasoning of domain information and knowledge, i.e., the specification of tasks. In other words, operators in the domain space are also objects in the task space. They are associated and configured to create meaningful tasks. Other objects in the task space are entities used for the evolution of these tasks. Operators in the task space are meta-actions that manipulate tasks in the domain space. Also, meta-tasks (e.g., how to organize, staff, and plan primary tasks) in the task space are combinations of meta-actions.

The *strategy space* stores strategic knowledge which directs tasks of organizational work and task performance, such as control structures. Objects in the strategy space, just as in the tasks space, are meta-tasks defined in the task space and other associated entities. Operators in the strategy space are super-meta-actions,[3] which manipulate meta-tasks in order to determine their control structures.

In this representation, when an operator is applied to a state[4] of its space, it creates a new state. The application of an operator is a step in the application of a task which consists of a set of ordered operators. In terms of the three spaces, *task execution* is a reasoning process for how to apply an operational specification (an object in the task space), which is a task in the domain space, on (the state of) the domain space to infer to a new state through continuous application of operators according to specified plans. *Meta-reasoning* is a process of applying a strategic specification (an object in the strategy space) on a state of the task space to infer to a new state through continuous application of operators according to the strategic specification. Two types of task performance skill are modeled: individual task performance and collective task performance.

Individual task performance models the agent's ability to perform tasks individually. It is conceptualized as a combination of reasoning and meta-reasoning processes

[3]Super as in superclass, class, subclass hierarchies.

[4]A *state* refers to a snapshot of interrelated object–attribute values in the web. Thus, a new state represents updates of object–attribute values or relations in the current state.

in all three spaces. When a problem is presented, the agent first chooses a strategy to deal with it. Meta-reasoning is then performed to create a meta-task for the problem, which in turn can be used to produce a copying action on a problem-solving method, i.e., a task in the task space. Next, the task is performed in the domain space to produce a resolution. When a solution or a task for a problem is known and available, reasoning is the only process issued to create the solution.

Collective task performance skill refers to an agent's ability to work with other agents through interactions to get things done jointly. This collective intelligence, based on individual task performance, supports three basic kinds of interaction: communication, synchronization, and articulation. *Communication among agents* is a way to exchange information. In communication, agents exchange their knowledge about the web of resources and situations by sending and receiving messages. Through message exchange, they can transfer their manipulation power. They can also integrate individual efforts by combining exchanged individual products together. *Synchronization among agents* arranges schedules for a group of agents to come together in order to perform collective tasks. For a collective task, all its performers have to be present for them to be executed. On the other hand, these agents are normally executing their own individual tasks while some of them may initialize collective tasks at any moment. Synchronization arranges schedules for collective tasks and is responsible for the followup actions when this fails. *Articulation*, at last, handles unexpected events which stop normal task performance. Articulation is the way to amplify individual skill and intelligence in a workplace [1], [11], [13], [18], [24] and is discussed in [18].

V. Behavioral Simulation

With the specification of a software development process model and a set of agents, *behavioral simulation* is defined as the agent's symbolic execution of task specification using available resources over a period of time. The trajectory of this symbolic execution is recorded as the predicted development history of the process and the behavior of the agents is exhibited through their simulated task performance.

The behavioral simulation generally begins with a set of agents with their behavioral specification defined according to their current task performance knowledge and skill. These agents are given a set of tasks as their assignment. A set of resources is also provided along with the tasks. Some of these resources will be consumed during the task performance. Others will only be used and returned (i.e., reusable resources). All these objects are specified within a single state as the initial instance of a process model as the starting position. A *state* is used to model the trajectory at a different instance in time. Each state is created by actions, and actions are linked as tasks. Fig. 4 suggests a description of this behavior simulation. In the picture, Time$N$ and Time$N+1$ are states and lines represent actions. Actions use some resources which are represented as circles. Overlapped circles indicate resource requirement conflicts, which can be resolved either through synchronization or articulation [18].

There are several requirements to be observed during the behavioral simulation.

1) The task assignments are stable, if not otherwise explicitly changed. This is to say that all agents must finish their assigned tasks before the simulation is done, unless situations emerge which prevent or delay their completion.

2) At any time instance, an agent cannot perform two actions simultaneously. But, an agent may perform several tasks concurrently during a period.

3) At any time instance, an agent cannot work on more tasks than its available supply of resources allow.

4) The preconditions associated with an action must be satisfied before it can be executed. These preconditions include resource requirements, partial ordering of execution, and execution authorization.

The behavioral simulation starts from the initial state and simulates the agents' activities performing their assigned tasks. Simulation of task performance is accomplished by execution of the task specification in top-down fashion. The higher levels of a task hierarchy provide information about work assignment and resource allocation. The lower levels of the task hierarchy provide links to procedural definitions which are executable in order to create new states. At each simulation step, symbolic execution is done by first propagating necessary information from high levels to lower ones, checking preconditions of an action, and then invoking the associated procedural definition to propagate changes. All the changes from different agents are then combined to form a new state of the software process model. Two things condition the new state and the task execution. First, each agent selects the executing action out of its own choice. The selected action is among its current work assignments. The choice can be influenced but not determined by outsiders. Second, there are probably conflicts in resource requirements and the changes which need to be solved through articulation [18]. Part of the symbolic execution of an action is shown in Fig. 5 where the rule gets an agent and its agenda, finds its current action, and starts to execute the action symbolically.

The final result of the behavioral simulation is a trajectory over a period of time, in which every action of the tasks is performed once by a subset of the agents at a time instant. These trajectories can be made persistent and evolvable. Such trajectories can be subsequently studied according to different criteria in order to analyze information about task performance, such as the agents' behavior during execution, their productivity, resource utilization, alternative "what-if" scenarios, and other interesting properties. However, the task execution is not guaranteed to finish successfully. Problems may rise due to unexpected events that need to be *articulated*. Articulation of task performance also affects the above criteria and their consequences can be tracked as well [18].

Based on the Articulator meta-model, many types of

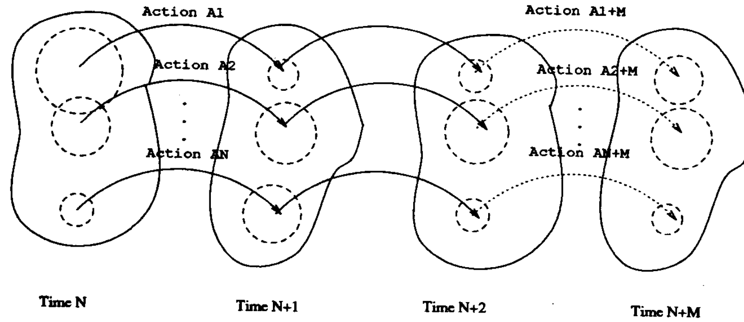


Fig. 4. The behavioral simulation of agents.

```
;; run the current action. The rule selects the agent, its agenda, and
;; its current action as the conditions. It then starts execution

(p Run-action :context t
  (INDIVIDUAL-AGENT
     ^instance <> ()
     ^schema-name <agent>
     ^controller-goal Controller-action-checked
     ^agent-has-agenda <agenda>
  )
  (AGENDA
     ^instance <> ()
     ^schema-name <agenda>
     ^current-slot <c-slot>
     ^time-slot-allocation <slots>
  )
  (ACTION
     ^instance <> ()
     ^schema-name <action>
     ^schema-name (select-current-action <> <slots> <c-slot>)
  )
  -->
  (format t ''Start to execute action ^A^&'' $<action>)
  (new-value $<agent> 'controller-parameter1 (current-p-1 $<slots> $<c-slot>))
  (new-value $<agent> 'controller-goal 'run-action))
```

Fig. 5. An SPSL rule for an individual-agent action.

# of Agents	*# of Tasks*	*# of Actions*	*Agent-Task*	*Agent-Action*	*Communication*
One	One	Many	1 – 1	1 – 1	None
One	Many	Many	1 – N	1 – 1	Inter-task Scheduling for One Agent
Many	Many	Many	1 – N	1 – 1	No Communication among the Agents
Many	One	Many	M – 1	1 – 1	Synchronous Combination of Results
Many	One	Many	M – 1	M – 1	Synchronization of Performance
Many	Many	Many	M – N	M – 1	All the Above

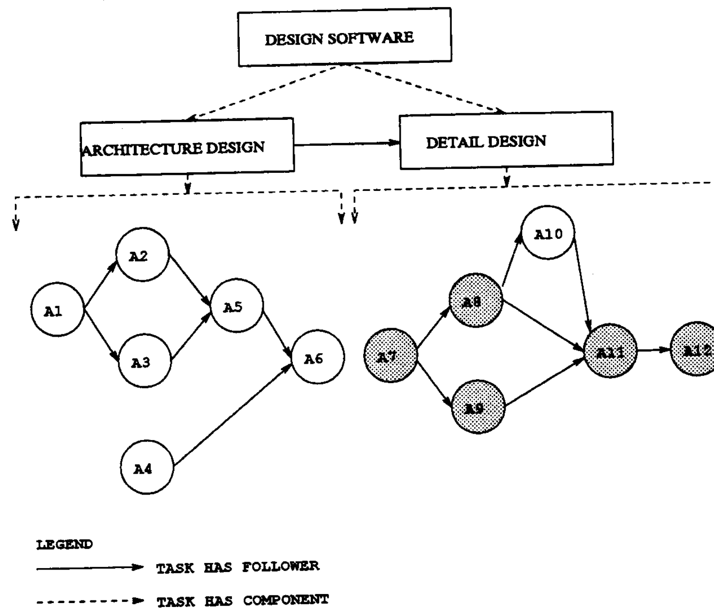
Fig. 6. Types of task performance by agents.

agents and tasks can be assigned to a software development process. This means that behavioral simulation can be divided into several types according to the number of involved agents, tasks, and the communication patterns between the agents shown in Fig. 6.

Let us consider our earlier example of Team-A (Fig. 3) again. The example is input into the Articulator and then simulated by the behavior simulator. In the example, there are two agents performing task "Design FOO." The task specification and work assignment are shown in Fig. 7. The resource requirements of Design-FOO are also given, but we can only show a single requirement here as in Fig. 8. In addition, Mary has another task, and thus sends a message to Peter, another member in company F, for assistance.

Due to space limits, we only present a summary report in Fig. 9 of the simulation here, which is obtained from the trajectory history and provides condensed information about the simulation. Then we discuss the types of behavior demonstrated in this example.

This behavioral simulation involves multiple agents performing a single task that requires the combination of




```
ACTIONS:

A1:  Establish system structure      A2:  Decompose system
A3:  Establish subsystem interface   A4:  Inform Joe about the task
A5:  Document architecture design    A6:  Validate architecture design
A7:  Design module structure         A8:  Develop data representation
A9:  Detail subsystem interface      A10: Design system interface
A11: Define algorithm                A12: Validate detail design
```

Fig. 7. Task decomposition of design-FOO. The unshaded actions (indicated by circles) are assigned to Mary and the shaded actions are assigned to Joe.

```
;; System-data-structure-spec is a resource manipulated in Design-FOO.
;; It is created by Developing-data-representation, and used in
;; Defining-algorithm and Designing-system-interface.
(define-object System-data-structure-spec
  (is-a DOCUMENT)
  (resource-required-by-task-chain
     Defining-algorithm Designing-system-interface)
  (resource-provided-by-task-chain
     Developing-data-representation)
  (resource-being-used-in-task-chain
     Defining-algorithm Designing-system-interface))
```

Fig. 8. A resource specification by SPSL.

each agent's task results. Initially, it has three agents and two tasks. However, our example focuses on one which is performed by two agents in combination. During the performance, a task–action ordering emerges. When either of two actions can be executed at the same time, an agent selects one randomly.

The agents communicate twice in the simulation. At time 3, Mary sends a work assignment to Joe, who reads the message at time 5 and begins to perform the task at time 7. Mary sends a file to Peter at time 9, who reads it at time 12.

Lack of resources occurs twice, and both are resolved through *accommodation*. At time 7, when Joe tries to start his task execution, the `Valid-document-spec`, a document created by action `Validating-architecture-design`, does not exist at the moment. Since Joe chooses a waiting strategy to accommodate (Fig. 3), he simply waits for the resource. Fortunately, the resource becomes available at time 8, so he continues. At time 8, Mary encounters the same problem. She prefers to switch to another task as her accommodation strategy, so she selects to perform another task: `Send-file-to-peter` and resume the original task at time 10, when the resource is available.

The task execution completes in 11 time steps by two agents. In total, there are 22 time steps from two agents of which 12 are used to perform the task, 1 for `Send-file-to-peter`, 1 for waiting, and 1 for switching. The other time steps ("slack time") could be utilized for other task performance if needed.

VI. The Query Mechanism

The query mechanism accepts user queries to retrieve information from the Articulator meta-model, the software process models, and their instances.

The query functions are built in bottom-up fashion. A

TIME	*MARY*	*JOE*	*PETER*
1	IDLE	IDLE	IDLE
2	ESTABLISH-SYSTEM-STRUCTURE	IDLE	IDLE
3	INFORM-JOE	IDLE	IDLE
4	ESTABLISH-SUBSYSTEM-INTERFACE	IDLE	IDLE
5	DECOMPOSE-SYSTEM	READ-MAIL	IDLE
6	DOCUMENT-ARCHITECTURE-DESIGN	IDLE	IDLE
7	VALIDATE-ARCHITECTURE-DESIGN	ACCOMMODATE (WAIT)	IDLE
8	ACCOMMODATE (SWITCH)	DESIGN-MODULE-STRUCTURE	IDLE
9	SEND-FILE-TO-PETER	DEVELOP-DATA-STRUCTURE	IDLE
10	DESIGN-SYSTEM-INTERFACE	DETAIL-SUBSYSTEM-INTERFACE	IDLE
11	IDLE	DEFINE-ALGORITHM	IDLE
12	IDLE	VALIDATE-DETAIL-DESIGN	READ-MAIL
13	IDLE	IDLE	IDLE

Fig. 9. Summary of simulation result.

```
;; Definition of meta-knowledge schema in SPSL with definitions for explanation.
(define-object Meta-knowledge
    (is-a SCHEMA)
    (definition)
    (methods-and-procedures)
    (reason-or-explanation)
    (unit)
    (literature-available))

;; Definition of meta knowledge for RESOURCE
(define-object Meta-resource
    (is-a META-KNOWLEDGE)
    (definition "RESOURCE is the basic entity in KB. It provides basic
            descriptions about entities in the meta-model. Every object
            must be a class of RESOURCE or an instance of RESOURCE.")
    (methods-and-procedures "A resource can be created, used, and consumed")
    (reason-or-explanation NA)
    (unit NA)
    (literature-available "mi"))

(attach-meta-schema 'RESOURCE 'Meta-resource)

;; Meta-knowledge query - WHAT question:
((q-WHAT ?ENTITY ?DEFINITION) ;
    (:RELATED ?ENTITY IS-A RESOURCE)
    (BIND ?DEFINITION (GET-VALUE (GET-META-SCHEMA ?ENTITY) 'DEFINITION)) !)

((q-WHAT ?ENTITY ?DEFINITION) ;
    (= ?DEFINITION "There is no such an entity in KB") ! fail)

;; Example of use of WHAT question: what is Company-F?
(q-WHAT Company-F ?DEFINITION)
    "Company-F is a software vendor. It develops software on SUN systems.
    Currently it has three members: Mary, Joe and Peter."
```

Fig. 10. Meta-knowledge in SPSL and its queries.

set of atomic functions, implemented as forward-chaining rules, are used to get very basic information about attributes and relations. They are atomic because they only retrieve attributes, relations, and their values. Higher level functions involve the knowledge representation and provide information about the representation. Users are encouraged to develop their own queries using the facilities we provide. The main concern in the query mechanism is to provide a set of functionally-complete basic facilities.

There are four types of queries supported in the query mechanism: meta-knowledge queries, information queries, history queries, and what-if queries.

A *meta-knowledge query* provides the definition of an entity and its related terminology in the Articulator. It is based on the information given when an object is defined, which is stored as meta-knowledge. An example definition of meta-knowledge appears in Fig. 10. This function is intended to help new users to understand the Articulator meta-model. Users can also provide their own meta-knowledge for models they defined, in order to help guide other users' interactions with a given model. A meta-knowledge query is in form of `q-WHAT`. For example, Fig. 10 also lists a meta-knowledge query about company F and the question is shown in the figure.

An *information query* provides information about either a state of a software process model or the model itself. It

```
;; IS question: Is Mary in the meta-model?
(q-is Mary)
     true

;; RELATION question: How are Mary and Joe related?
(q-RELATION Mary Joe)
     Mary individual-in-collective-agent Company-F collective-agent-has-member Joe.
;; It means Mary and Joe are both in Company-F

;; FOLLOWER question: What are the followers of Develop-data-representation?
(q-follower Develop-data-representation ?what)
     ?what = (Design-system-interface Define-algorithm Validating-detail-design)
```

Fig. 11. Examples of information query.

is generally concerned with resource values and configurations. Typical questions answered include "Is Peter a member of Team-A at time 1?," "What are the relations linking Peter and Mary now?," etc. An information query has several basic functions for this kind of deductive retrieval. For example, `q-is` checks the existence of an entity in the status, and `q-relation` finds the relations which link two given entities. Through the use of an information query, every value and every relation within a state can be retrieved without difficulty.

More complicated queries have been implemented as examples of query building using these basic functions. For example, `q-follower` and `q-predecessor` are used to find follower actions and predecessor actions of a given action along relation `task-force-has-follower`. These two queries are useful for users to check the configuration of the tasks they perform. They are implemented by the `q-relation` query using `task-force-has-follower` as the given relation. Another example is to get all component modules of a given software project, implemented as `q-soft-configuration`. Many such queries with specific requirements can be built in the same manner. In Fig. 11, we provide some information queries about our simple model of company F.

A *history query* traverses a trajectory of states created in a simulation, collects a record of changes on specified entities, then summarizes them to give clear and condensed information about these changes. Typical information provided in history queries includes the activities performed by the agents in the simulation period and the resources consumed or produced by agents. Other specific queries may ask about the consequence of a particular action, a value change, or an inserted relation.

Implementation of the history query is based on information queries and the instantiation manager. The former provides facilities to retrieve information within a state, while the latter gives the capability to traverse within the state trajectory. Also a history query has a facility to sum up gathered information. The simulation result presented in Fig. 9 comes from a history query.

A *what-if query* includes a combination of simulation and history queries. It starts from a given state, or a modification of a state in the middle of a sequence trajectory, and calls the behavior simulator to simulate the given updated scenario. When the simulation is done, a history query is activated to gather required information. What-if queries are designed to facilitate testing of the hypothesis scenario and handling of unexpected events.

VII. Conclusion

Within the Articulator project, we propose some novel contributions to the study of software engineering processes using a knowledge engineering environment. We create a tractable open-system model of software processes and resource infrastructures that are articulated by agents working in development settings. We explore relationships among the components, such as software processes, development resources, and developers, within the model and their impact on the software development products, processes, and workplace settings under study. We also provide formalisms to represent task performance skill. We present a meta-model of software processes which is suitable for describing software process models. All these contributions are further enhanced through the simulation of the dynamics of software process models as a basis for querying the state of values in the represented model, the simulated trajectory, and the recorded process history. As the Articulator becomes more complete, we hope to provide a framework to further assist the interactive empirical study of large scale software development projects.

References

[1] S. Bendifallah and W. Scacchi, "Understanding software maintenance work," *IEEE Trans. Software Eng.*, vol. 13, no. 3, pp. 311-323, Mar. 1987.

[2] —, "Work structures and shifts: An empirical analysis of cooperative work in software specification," in *Proc. 11th Int. Conf. Software Eng.*, Pittsburgh, PA, May 1989, pp. 260-270.

[3] Carnegie Group Inc., *KnowledgeCraft™ User's Guide, Vol. 1, Vol. 2, and Vol. 3*, 1986.

[4] B. Curtis, H. Krasner, V. Shen, and N. Iscoe, "On building software process models under the lamppost," in *Proc. 9th Int. Conf. Software Eng.*, 1987, pp. 96-103.

[5] B. Curtis, H. Krasner, and N. Iscoe, "A field study of the software design process for large systems," *Commun. ACM*, vol. 31, no. 11, pp. 1268-1287, 1988.

[6] E. H. Durfee, V. R. Lesser, and D. D. Corkill, "Cooperation through communication in a distributed problem solving network," in *Distributed Artificial Intelligence*, M. N. Huhns, Ed. Los Altos, CA: Morgan Kaufmann, 1987, pp. 29-58.

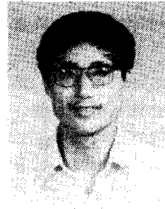
[7] J. J. Elam, D. B. Walz, H. Krasner, and B. Curtis, "A methodology for studying software design teams: An investigation of conflict behavior in the requirements definition phase," in *Empirical Studies of Programmers (Second Workshop)*, G. M. Olson, S. Sheppard, and E. Soloway, Eds. Ablex, 1987, pp. 83-99.

[8] P. K. Garg and W. Scacchi, "ISHYS: Designing an intelligent software hypertext system," *IEEE Expert*, vol. 4, no. 3, pp. 52-63, 1989.
[9] L. Gasser, "The integration of computing and routine work," *ACM Trans. Office Inform. Syst.*, vol. 4, no. 3, pp. 205-225, July 1986.
[10] M. Genesereth, "An overview of meta-level architecture," in *Proc. AAAI-83*, Washington, DC, 1983.
[11] E. M. Gerson and S. L. Star, "Analyzing due process in the workplace," *ACM Trans. Office Inform. Syst.*, vol. 4, no. 3, pp. 257-270, July 1986.
[12] R. Guindon, H. Krasner, and B. Curtis, "Breakdowns and processes during the early activities of software design by professionals," in *Empirical Studies of Programmers (Second Workshop)*, G. M. Olson, S. Sheppard, and E. Soloway, Eds. New Haven, CT: Ablex, 1987, pp. 65-82.
[13] C. Hewitt, "Offices are open systems," *ACM Trans. Office Inform. Syst.*, vol. 4, no. 3, pp. 271-287, July 1986.
[14] K. E. Huff and V. R. Lesser, "A plan-based intelligent assistant that supports the software development process," *ACM Software Eng. Notes*, vol. 13, no. 5, pp. 97-106, Nov. 1988.
[15] W. S. Humphrey and M. I. Kellner, "Software process modeling: Principles of entity process models," in *Proc. 11th Int. Conf. Software Eng.*, Pittsburgh, PA, May 1989, pp. 331-342.
[16] G. Kaiser, P. Feiler, and S. Popovich, "Intelligent assistance for software development and maintenance," *IEEE Software*, vol. 5, no. 3, 1988.
[17] R. Kling and W. Scacchi, "The web of computing: Computer technology as social organization," *Advances Comput.*, vol. 21, pp. 3-91, 1982.
[18] P. Mi and W. Scacchi, "Negotiation: A collective problem-solving approach," Working Paper, SF-89-03, Comput. Sci. Dep., U.S.C., Los Angeles, CA, June 1989.
[19] L. Osterweil, "Software processes are software too," in *Proc. 9th Int. Conf. Software Eng.*, 1987, pp. 2-13.
[20] A. Sathi, M. S. Fox, and M. Greenberg, "Representation of activity knowledge for project management," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-7, pp. 531-552, 1985.
[21] W. Scacchi, "Managing software engineering projects: A social analysis," *IEEE Trans. Software Eng.*, vol. 10, no. 1, pp. 49-59, Jan. 1984.
[22] —, "The USC System Factory Project," in *Proc. Software Symp. '88*, (Keynote Address), Software Engineers Association, Tokyo, Japan, June 1988, pp. 9-41.
[23] —, "Models of software evolution: Life cycle and process," Tech. Rep. CM-10-87, Software Engineering Institute, Carnegie-Mellon Univ., Pittsburgh, PA, 1987.
[24] A. Strauss, "The articulation of project work: An organizational process," *Sociological Quarterly*, vol. 29, no. 2, pp. 163-178, 1988.
[25] M. Stefik, "Planning and meta-planning (MOLGEN: Part 2)," *Artificial Intell.*, vol. 16, no. 2, pp. 141-169, May 1981.
[26] J. C. Wileden, "This is IT: A meta-model of the software process," *ACM SIGSOFT Software Eng. Notes*, vol. 11, no. 4, pp. 9-11, Aug. 1986.
[27] L. G. Williams, Software process modeling: A behavioral approach," in *Proc. 10th Int. Conf. Software Eng.*, 1988, pp. 174-200.

Peiwei Mi received the B.S. and M.S. degrees in computer science from the University of Science and Technology of China, in 1982 and 1984, respectively.

He is a Ph.D. student in the Computer Science Department, University of Southern California, Los Angeles. His research interests include knowledge-based systems supporting the software process, organization analysis of system development projects, and distributed problem solving.

Mr. Mi is a student member of the IEEE Computer Society.



Walt Scacchi (S'77–M'80) received the B.A. degree in mathematics, the B.S. degree in computer science in 1974 from California State University, Fullerton, and the Ph.D. degree in information and computer science from the University of California, Irvine, in 1981.

Since then, he has been on the faculty in the Computer Science Department, University of Southern California, Los Angeles. Since 1981, he created and directs the System Factory Project at USC, the only software factory research project in a U.S. university. His research interests include very large scale software engineering, knowledge-based systems supporting the software process, and organizational analysis of system development projects. He is an active researcher with more than 70 research publications, and numerous consulting and visiting scientist positions with firms including AT&T Bell Laboratories, Microelectronics and Computer Technology Corporation (MCC), the Software Engineering Institute at Carnegie Mellon University, and SUN Microsystems.

Dr. Scacchi is a member of the Association for Computing Machinery, AAAI, and the Society for the History of Technology (SHOT).