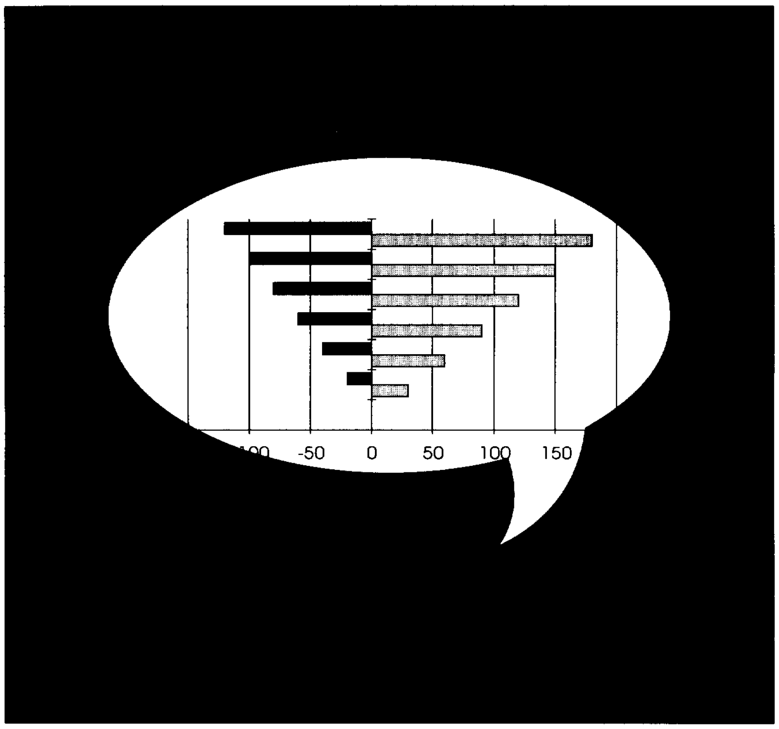
MODELING CONTROL IN RULE-BASED SYSTEMS

The Comex modeling tool addresses the rule-based paradigm's biggest shortcoming by adding control structures to executable models. Beginning with the specification phase, you can simulate a system's behavior, then map tasks to a rule base. The result is a structured, rule-based system built according to accepted software-engineering principles.

METTE VESTLI and INGE NORDBØ
Sintef Delab
ARNE SØLVBERG
University of Trondheim, NTH



During the 1980s, the rule-based paradigm, which describes the world in terms of conditions and their consequences, was promoted as the ultimate solution to the problem of engineering reliable, maintainable software. Later, when it became obvious that the rule-based paradigm could not fulfill these promises, attitudes toward it became very negative.

The problem was that production rules, which were intended to isolate both developers and maintainers from implementation details about control and data, actually isolated them from the procedural control structures and abstractions as well. All these factors seemed to grow exponentially as application size increased.

Xiaofeng Li summed up the major criticisms of the rule-based paradigm:[1] Rule-based systems, he wrote, do not simplify problems but merely shift complexity from one level to another. Moreover, they are difficult to test (and hence unreliable) because they lack modularity and abstraction facilities. And the representation obscures the application's structure, dataflow, and control structure, making rule-based systems difficult to maintain.

We believe this criticism is too harsh. Our experience is that the rule-based paradigm is very useful when it is used according to well-known software-engineering principles and is supported by modeling tools. The rule-based paradigm expresses some kinds of knowledge very well. For example, regulations, directions, and instructions are written in rule form, and decision-making knowledge is often

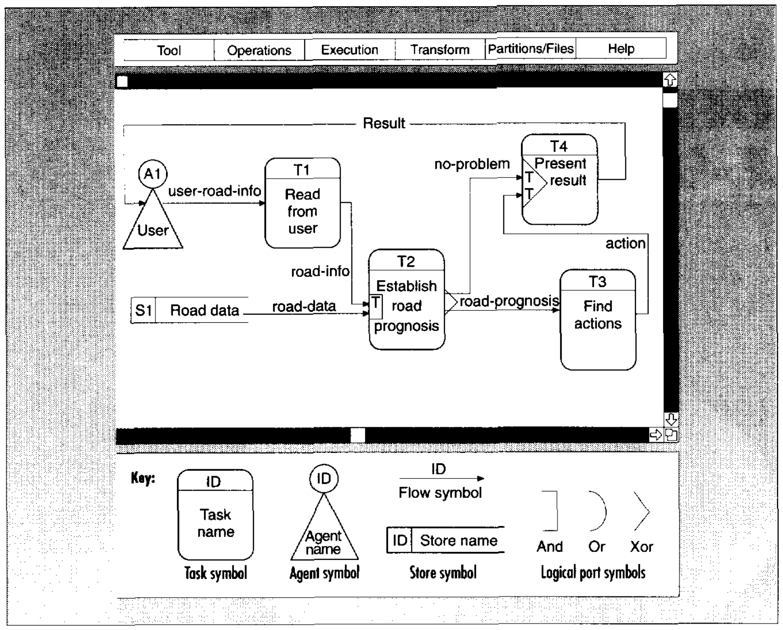


Figure 1. *Top-level task model for a road-maintenance system.*

easily expressed as rules. We should take advantage of the expressiveness of rules, and combine this paradigm with others to describe knowledge that is not easily expressed with rules. To a great degree, the shortcomings of the rule-based paradigm are manageable if you proceed methodically and use support tools.

We have developed such a tool, Comex (Control Knowledge Modeling and Execution Tool). Comex focuses on modeling a problem-solving strategy and representing *control knowledge*, the knowledge of how to select among several problem-solving actions. In addition, Comex has facilities that let you execute early versions of the model to simulate the intended system's behavior and continuously execute the evolving model, until it becomes the real system. As you gradually make the model more detailed, you assign problem-solving actions to dedicated rule bases or other system components. In this way, Comex supports interactive development and integrates different paradigms.

WE SHOULD COMBINE THE ADVANTAGES RULES HAVE WITH OTHER PARADIGMS.

We based Comex on the Phenomena, Processes and Programs environment for modeling information systems,[2] developed at the Norwegian Institute of Technology. PPP supports top-down, interactive development of all aspects of information-systems development, including process and data- and control-flow modeling. A PPP model may be transformed into a set of temporal logic rules,[3] so it can be evaluated by a rule manager.

Comex uses the same mechanisms, but to model knowledge-based systems.

MODELING PROBLEM-SOLVING STRATEGY

Comex supports top-down development. You use a graphical language to specify the system's problem-solving strategy as a hierarchical *task model*. You can then execute this model at any stage of decomposition to simulate the system's behavior. As you fill in the details, the model gradually becomes the real system.

For example, suppose you wanted to develop a knowledge-based decision-support system for winter road maintenance. The system should help highway supervisors determine in advance where ice and snow conditions may develop so that they can arrange for salting — which is more effective if applied before freezing starts — and snow removal. Among other things, the system is intended to help supervisors minimize the amount of salt used because salt is expensive and damaging to the environment.

Assume that measurement stations along the roads feed data into the system via sampling and that supervisors can get a graphical presentation of the data.

Typically, there are four main steps to iteratively develop a knowledge-based system.

1. Create and edit a task model. A task model specifies an application's structure and dynamics. It includes a description of the task, the information flow among tasks, and the control flow that determines the sequence of task execution. Tasks are decomposed until they can be solved by a single rule set. Each task at the lowest level has an action part that activates a procedure to carry out the task.

2. Add control knowledge. As you decompose the task model, you use symbols to add logical ports to the tasks and express conditions for their activation and termination.

3. Add object-knowledge reference. Object knowledge describes terms and relations within the application domain. It is usually acquired through knowledge-acquisition methods and tools and may be expressed as rules. In this step, you map a task to the object knowledge it needs.

4. Execute the task model. Comex uses the control specification and the mapping from tasks to object knowledge to simulate the application's behavior, printing an execution trace to give you an overview of how the model behaves in different situations. In early development phases, when little object knowledge has been acquired, Comex executes the empty model, prompting you for the dataflow values to use during execution. In this way, you can simulate the sequence in which tasks are carried out, depending on the input values and intermediate results.

The main cycle of iteration moves between modeling (the first three steps) and execution (the fourth step). Inside the modeling stage is another cycle of iterations, as you progressively create a task model and add control and object knowledge.

TASK MODEL

The first step is to construct a top-level task model, which shows the tasks needed to solve the problem, how they exchange information, and how they communicate with the external world.

Figure 1 shows a simple top-level model for the road-maintenance system. It involves four tasks, one agent, one store, and flows.

♦ *Tasks* express activities.

♦ *Agents* specify the interface to the external world — highway supervisors, in this case.

♦ *Stores* model internal data sources — sampled road data, in this case.

♦ *Flows* specify the exchange of data among tasks, agents, and stores. Flows are given unique names that are used by the execution facilities; to aid comprehension, flow names should also indicate content.

In Figure 1, Task T1 (read from user) receives a measurement-station identifier from the user. Task T2 (establish road prognosis) uses this input and input from a store of road-condition data to determine if there is no need for action (when the air temperature is too high, for example) or if task T3 (find action) should be activated. Task T4 presents to the user either a recommended action or the fact that no action is needed.

Because task T2 is too complicated to be solved by a single rule set, you must decompose it into subtasks. Figure 2 shows the decomposition. Task T2.In analyzes the data supplied by the measurement station. If it reports no ice or snow and no conditions likely to lead to ice or snow, the output is no-problem-situation.

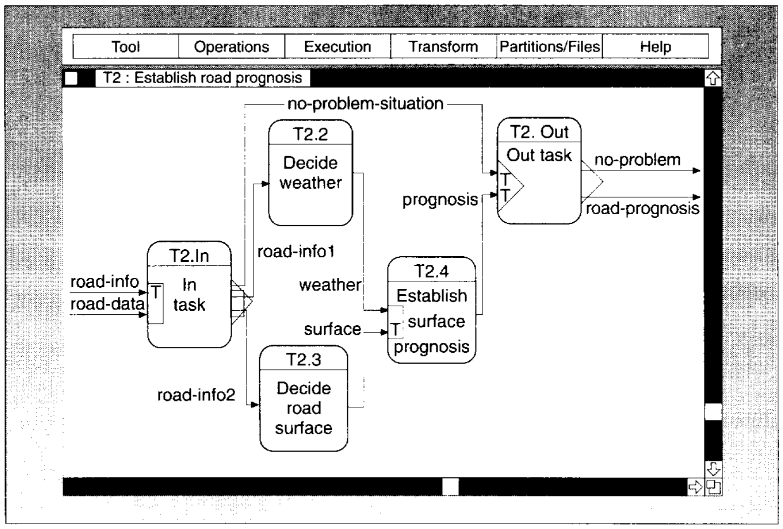


Figure 2. Task model for a subtask.

If the measurement station reports otherwise, task T2.In supplies parameters like rain, snow, and temperature to task T2.2 and parameters like road-surface temperature to task T2.3. Task T2.4 analyzes this data to make a prognosis. The tasks T2.In and T2.Out preserve consistency with the upper level model, and split and combine flow values.

CONTROL STRUCTURES REPLACE THE METARULES FOUND IN TRADITIONAL RULE-BASED SYSTEMS.

CONTROL KNOWLEDGE

In Comex, tasks are related to one another through flows, which represent control structures. The control structures replace the levels of metarules that represent control in traditional rule-based systems. Metarules — rules about how to use rules — let you reason about control to improve execution, but provide little help in understanding the system's behavior. In a Comex model, you represent control with flows and triggers attached to them and with ports, which represent a task's start and end conditions.

♦ *Trigger symbols.* When a flow is labeled with T, it may activate a task. If a flow is the only flow entering a task, it is a triggering flow by definition.

♦ *Logical ports* express the conditions that govern a task's activation and termination. There are three types: *and* ports require that all flows have a value or control signal before an activity can be performed; *or* ports require that at least one of the flows have a value; *xor* (exclusive or) ports require that exactly one flow have a value.

In Figure 1, task T1 activates (triggers) task T2. When activated, task T2 requires a set of road data. Furthermore, both tasks T2 and T3 can activate task T4, so you label both flows with a trigger and give task T4 an xor symbol. Hence, if the flow no-problem from T2 has a value, then the flow action from T3 cannot have a value, and vice versa.

In this way, flows and ports specify the control structure.

This explicit modeling of control knowledge lifts task sequence and action

```
NAME:      T2.4

LABEL:     ESTABLISH
           SURFACE
           PROGNOSIS

ACTION:    (set-flow-value 'prognosis
              (prove  '((exists x)
                          ((exists y)
                          (and    (mst x)
                                  (surface-prognosis x y ))))))

USES:      SURFACE-PROGNOSIS-RULES

IN-PORT:   (and weather surface)

OUT-PORT:  prognosis
```

Figure 3. Task description.

```
Execution:
Task T1 outflow:        road-info: mst-52
Task T2.in outflow:     no-problem-situation: nil
                        roadinfo1:((weather-temp-prognosis mst-52 1)
                                   (surface-now mst-52 wet))
                        roadinfo2:((weather-temp-prognosis mst-52 1)
                                   (surface-now mst-52 wet))
Task T2.2 outflow:      weather: (weather-prognosis mst-52 rain)
Task T2.3 outflow:      surface: (road-temp-prognosis mst-52 -1)
Task T2.4 outflow:      prognosis: (road-prognosis mst-52 ice)
Task T2.out outflow:    road-prognosis: (road-prognosis A3 ice)
                        no-problem: nil
Task T3 outflow:        action: ( (action A3 salting)
                                  (salting-type wet-salt)
                                  (concentration 5 g/m2))
Task T4 outflow:        result: "Start salting on A3 with wet salt,
                                 concentration 5 g/m2"
```

Figure 4. Execution trace.

choice out of the problem-solving knowledge bases. The control structure imposes constraints on each task's I/O, so it is easy to focus testing and maintenance on individual partial knowledge bases.

OBJECT KNOWLEDGE

In Figure 1, tasks T1 and T4 are only dialogues with the user and so contain no rule bases. Tasks T2 and T3, however, use rules to establish a prognosis and find an action. The subtasks T2.2, T2.3, and T2.4 in Figure 2 also use rules for reasoning.

It is in the action part of a task description that you specify which part of the object knowledge it should use, where to find it, and how to use it. For example, Figure 3 shows the task description for T2.4. The description lists the task's name and label, the action, the rule partition used, and the logical port expressions.

In the current implementation, the action is expressed in Lisp using a special-purpose editor. In this task, the action calls the expression `prove`, which does backward chaining on the rule partition. The `x` represents the measurement station; `y` the derived surface prognosis. The expression `set-flow-value` puts the result of the proof on the flow prognosis.

As Figure 3 shows, this task uses a rule from the surface-prognosis-rules rule base. This rule expresses the connections among three other rules: weather prognosis, surface condition, and surface temperature (the rules in this application are represented in first-order predicate logic):

```
((all x) (all y) (=>
   (& (weather-prognosis x rain)
      (surface-now x wet)
      (road-temp-prognosis x y)
      (less-than y 0))
   (surface-prognosis x ice)))
```

Again, `x` represents the measurement station being investigated. The rule says the prognosis for the surface is ice if the weather prognosis is rain, the surface is now wet, and the prognosis for the road temperature is below zero centigrade.

We organized the object knowledge according to the task decomposition, as a set of separate rule partitions activated individually:

```
Rule-partition
  |- Prognosis-rules
  |  |- Weather-rules
  |  |- Road-surface-rules
  |  |- Surface-prognosis-rules
  |- Find-action-rules
```

The control structure handles the flow of intermediate conclusions among rule partitions by sending values between tasks at runtime.

Because tasks represent a single problem-solving action, you can represent object knowledge in different formats, including rules, frames, and predicate logics. You can also easily integrate standard database calls or calls to numeric software packages. Hence, the task model can combine results obtained from reasoning in different knowledge bases and results from other software modules. This specification structure lets you cope with complexity.

Keeping the object knowledge in separate partitions and expressing the control knowledge in the task model keep the application's structure and control explicit. This makes knowledge-base testing and maintenance much easier. In addition, because each task is a limited problem, you can test and verify it individually until you achieve a sufficient level of reliability. And you can simulate the behavior of the complete system in the modeling tool by composing a model of pretested components.

MODEL EXECUTION

The Comex task-model executor uses the task model, the control specification, and task actions to execute the complete application. You provide input values, and the executor uses task actions to derive other flow values.

When it activates a task, the executor evaluates the input port. When the port is satisfied, it executes the task's action part. The action usually initiates reasoning in a knowledge base, but it may also initiate a call to a database or some other software module.

During task execution, output flows receive values. When the task's action is finished, the executor evaluates the output port to check if the termination condition is satisfied. It derives the next task to activate from the triggering conditions on the output flows.

In our example, we start the prognosis module by providing a measurement-station identifier. Task T1 passes this information to task T2, which fetches data from the store S1. When the situation requires salting, the executor prints a trace like the one in Figure 4. Had salting not been necessary, the output for roadinfo1 and roadinfo2 would have been nil.

When the model behaves as expected, you import it into the target environment. There it can serve as a specification for conversion to another formalism or be used directly if the target environment runs Comex. (Transformation issues such as preserving structure and modularity to keep a well-structured, well-documented system and knowledge semantics are beyond the scope of this article.)

Because it relies on traditional software-engineering principles, Comex produces well-structured systems that will answer doubts about whether the rule-based approach is practical. Comex scales up as well as traditional software engineering scales up, so it is useful for both large- and small-system development

In the context of rule-based programming, the executable task model explicitly describes the tasks the system will perform and their activation sequence. The overall problem-solving strategy is expressed in terms of action sequences, displayed graphically.

Furthermore, Comex lets you combine rule-based reasoning with other paradigms. Because knowledge-based systems are usually modules within larger information systems, using this kind of modeling technique lets you integrate different kinds of modules in one model: Although their internal representations may differ, you can model the control structures among knowledge-based and other component types in the same way.

Comex is one of many tools included in the Knowledge Engineering Workbench developed in the ESPRIT Acknowledge project,[4] which supports the acquisition, modeling, and evaluation of knowledge. It is also included in a platform supporting uncertain, incomplete, and temporal-dependent applications, which is being developed by ESPRIT's Unite project.[5]

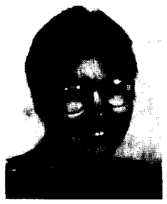
As part of the Knowledge Engineering Workbench, Comex has recently been used to develop PrintHelp, a knowledge-based help-desk application for Norway's largest insurance company,[6] and Dekspert,[7] a decision-support system for the Norwegian Public Roads Administration. Dekspert's runtime version is implemented in Neuron Data's Nexpert Object environment. ♦

ACKNOWLEDGMENTS

This work was conducted in the ESPRIT project Acknowledge. We thank the members of the Knowledge Engineering Lab at Sintef Delab and the Information Systems Group at the Norwegian Institute of Technology, University of Trondheim, for discussing and commenting on this article.

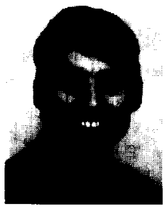
REFERENCES

1. Xiaofeng Li, "What's So Bad About Rule-Based Programming," *IEEE Software*, Sept. 1991, pp. 103, 105.
2. J.A. Gulla, O.I. Lindland, and G. Willumsen, "PPP — An Integrated CASE Environment," *Proc. Third Int'l Conf. CAiSE*, Springer-Verlag, Berlin, 1991, pp. 194-221.
3. J. Krogstie et al., "Information Systems Development Using a Combination of Process and Rule Based Approaches," *Proc. Third Int'l Conf. CAiSE*, Springer-Verlag, Berlin, 1991, pp. 319-335.
4. J.-Ch. Marty et al., "ACKnowledge: An Intelligent Workbench for the Knowledge Engineer," *Proc. Expert Systems World Congress*, Pergamon Press, New York, 1991, pp. 789-796
5. F. Ramparany et al., "Knowledge Integration and Interchange Issues in Applications Handling Imperfect and Time-Dependent Information," *Proc. IJCAI-93 Workshop Knowledge Sharing and Information Exchange*, Morgan Kaufmann, San Mateo, Calif., 1993, pp. 55-59.
6. O.J. Mengshoel, I. Nordbo, and I. Solvberg, "The Knowledge Engineering Workbench: Experiences from Building the Knowledge Base of a Helpdesk Application," *Proc. Int'l Conf. Artificial Intelligence, Expert Systems and Natural Language*, EC2, Paris, 1992, pp. 53-65.
7. T. Haugødegård et al., "Norwegian Public Roads Administration: A Complete Pavement Management System in Operation," *Proc. Int'l Conf. Managing Pavements*, Transportation Research Board, Washington, DC, 1994, to appear.



Mette Vestli is a research scientist in the Knowledge-Engineering Lab at Sintef Delab. Her interests are methods and tools for software engineering and knowledge engineering, including corporate knowledge.

Vestli received a degree in computer science from the University of Trondheim, NTH.



Inge Nordbø is a research scientist in the Knowledge-Engineering Lab at Sintef Delab. His interests are methods and tools for knowledge engineering, elicitation, modeling, and representation.

Nordbo received a degree in computer science from the University of Trondheim, NTH.

Arne Sølvberg is a coauthor of another article in this issue. His biography appears on p. 49.

Address questions about this article to Vestli or Nordbo at Sintef Delab, N-7034 Trondheim, Norway; mette.vestli@delab.sintef.no or inge.nordbo@delab.sintef.no, or to Sølvberg at Division of Computer Science, Norwegian Institute of Technology, N-7034 Trondheim, Norway; arne.solvberg@idt.unit.no.