Automatic Translation of Process Modeling Formalisms

George T. Heineman[1]
Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
heineman@cs.columbia.edu
TR CUCS-036-93
November 10, 1993

Abstract

This paper demonstrates that the enaction of a software process can be separated from the formalism in which the process is modeled. To show this, we present a means of automatically translating a process' specification, modeled using the STATEMATE approach, into an active environment, using the MARVEL system, which then supports the enaction of the process.

key words: Software process modeling, process enaction, formalism interoperability.

1 Introduction

The software community has recognized the importance of understanding and modeling the processes which are used to develop and support software. There are many objectives which have been cited as motivation for the development and application of software process models [8]. These include support for: automated execution and control, human interaction, managerial responsibilities, process understanding, and analysis of process. It is our belief that the actual enaction of a process can be orthogonal to the formalism in which the process is modeled. In many ways, this is true today, although mainly for unsatisfactory reasons. Many corporations have detailed processes on paper, which must be manually carried out by the individual users of the process. In most cases, there is no way to monitor or control the process, and the managers must simply hope that all the users follow the process.

This paper presents a method by which a process modeled using the SEI software process modeling approach [7] can be automatically translated into an active MARVEL [3] environment which can be used to enact the process. Section 2 of this paper describes the SEI software process modeling approach and partially reproduces Marc Kellner's

[1]Heineman is supported in part by IBM Canada

solution to the ISPW-6 standard benchmark problem. Section 3 describes the MARVEL approach to software modeling. Section 4 describes the compiler which translates the STATEMATE[2] specification of the process into a MARVEL environment. Section 5 outlines some future research directions.

2 SEI software modeling approach

The SEI approach [7] models the software process through the use of three interrelated perspectives which determine the who, what, where, when, and how of the process. The STATEMATE tool [2] is used to construct these perspective charts.

- Functional perspective – determines what the tasks are, and the information flow between the tasks
- Behavioral perspective – determines when and how tasks are performed
- Organizational perspective – determines who in the organization performs the tasks, and where the tasks are done.

Activity charts are used to model the functional perspective of the process. These charts focus on the activities being performed and hide the actual details of how they are carried out. Statecharts are used to provide the behavioral perspective, and focus on the timing and ordering of the individual process steps. The final perspective is provided by module charts which describe the organizational units involved in the process and the physical communication channels used to transfer information between the activities. These charts are interrelated to form a unified view of the process. To assign responsibilities for each activity, the activity and module charts are connected. Sequencing of activities is determined by the connections between the activity and statecharts. Since this paper focuses on the compilation of statecharts, we now describe them in more detail.

2.1 Statecharts

Statecharts are a form of state transition diagrams where boxes represent states of the process and arrows represent transitions between these states. Each state can either be primitive or be further subdivided into substates. If the subdivision represents a concurrent decomposition, then the state is an AND-state and each substate is an orthogonal component, otherwise the state is an OR-state, and the decomposition further refines the state. When the system is in a given state S, an arrow between S and another state T causes a state transition when the event and conditions for the arrow are satisfied. If the system is in an AND-state, the system is simultaneously

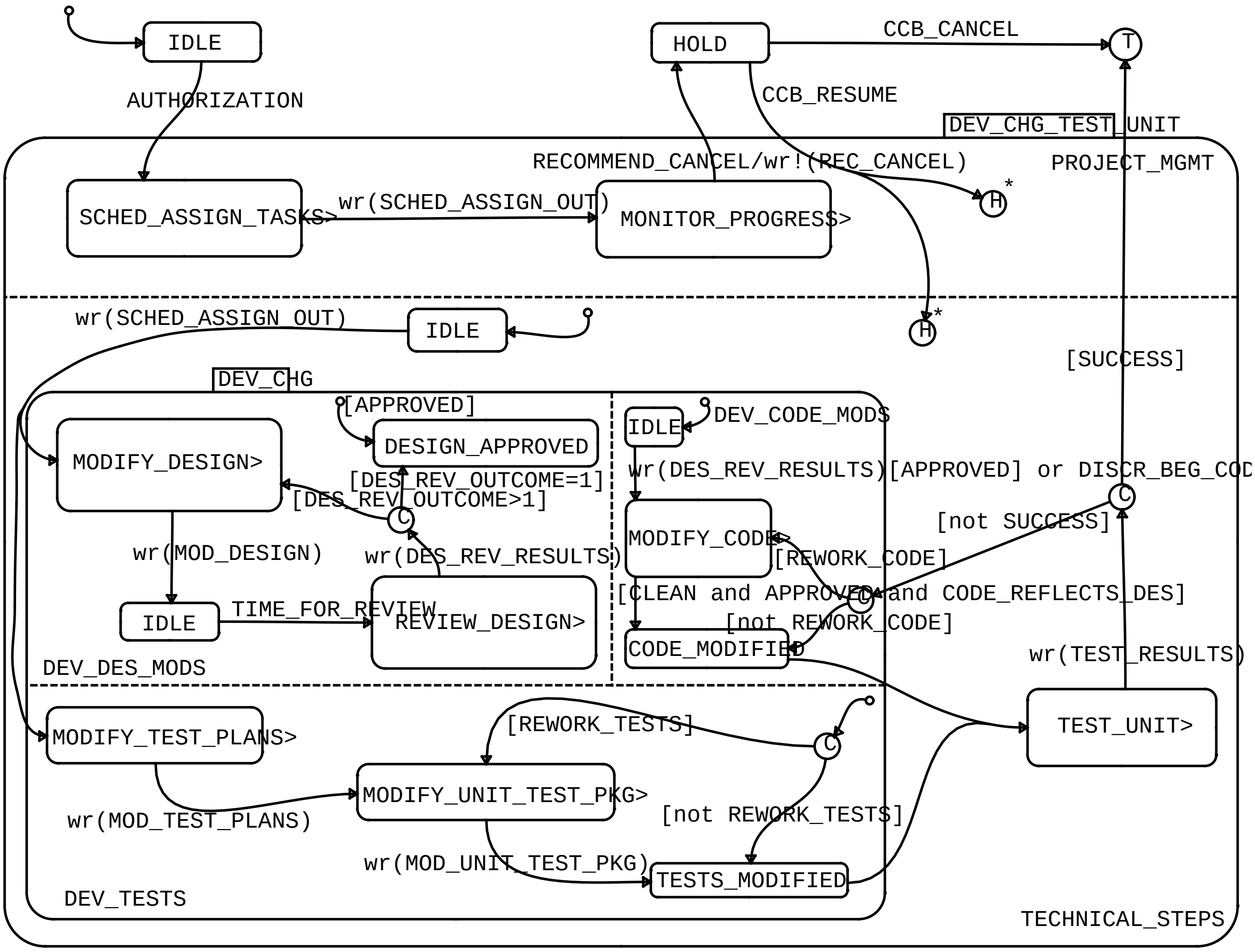

Figure 1: Sample Statechart

in a substate of each of its orthogonal subcomponents. Each transition can have a condition associated with it, which determines when it is valid; in addition, triggering events are associated with each transition to determine when it should be carried out. Transitions can have multiple sources, restricting it to occur only when the system is in each of the source states, or multiple targets, which activates a set of target states when the transition is made. Finally, statecharts can have history entities which maintain state information; this simplifies the writing of interrupts and special events.

Stepping through the statechart in figure 1 will clearly explain the statechart. The statechart is initialized in the IDLE state (default transitions have a small circle as their source). When the AUTHORIZATION event occurs, a transition is made to the SCHED_ASSIGN_TASKS state (and to PROJECT_MGMT since it is the parent). However, since PROJECT_MGMT is an orthogonal component of the DEV_CHG_TEST_UNIT AND-state (its name is in a box), the system must also simultaneously make a transition into the TECHNICAL_STEPS state, thus activating the default transition into

TECHNICAL_STEPS.IDLE. When the wr(SCHED_ASSIGN_OUT) event is generated, two transitions occur, first from SCHED_ASSIGN_TASKS to MONITOR_PROGRESS, and then from TECHNICAL_STEPS.IDLE to MODIFY_DESIGN and MODIFY_TEST_PLANS. Since DEV_CHG is an AND-state, the default transition into DEV_CODE_MODS.IDLE is taken.

When the RECOMMEND_CANCEL event occurs, the system makes a transition into the HOLD state. If, at this point, a transition were made back to MONITOR_PROGRESS, the state-information for the TECHNICAL_STEPS state would be lost, and the system would make a default transition back into TECHNICAL_STEPS.IDLE. To prevent this, on the CCB_RESUME event the system makes a transition to two deep history connectors (shown by H^*). A history connector belongs to a particular state, and a transition into one causes that state to revert to its previous configuration before the system had left it; in this example, PROJECT_MGMT would revert to MONITOR_PROGRESS, and TECHNICAL_STEPS to DEV_CHG. Since the history connector for TECHNICAL_STEPS is a deep history one, DEV_CHG also is directed to restore its state, so MODIFY_DESIGN, MODIFY_TEST_PLANS, and DEV_CODE_MODS.IDLE are made active again. If the history connector had not been deep, then the default transitions for DEV_DES_MODS, DEV_CODE_MODS, and DEV_TEST would have been followed. Finally, conditional transitions are made possible through the use of conditional connectors (C); conditions are surrounded by brackets, as in [SUCCESS]. This directs the transition to different targets based upon the logical value of the labeled conditions.

Statecharts can also have static-reactions associated with each state that can observe and influence the system. They are not transitions, sinc no state change occurs. Consider the MONITOR_PROGRESS state. This has two static reactions (not shown in the figure). The first is triggered when the state is entered:

```
entering/rd!(PROJ_PLANS)
```

This reads the project plans whenever this state is entered, ensuring that the most-recent project plans are available when monitoring the progress of the process. The second is more complicated:

```
TIME_FOR_REVIEW[in(MODIFY_DESIGN)]/wr!(REV_ASSIGNMENTS_DATES)
```

This synchronizes the two activities, and writes out the required changes to the assignments and dates when the TIME_FOR_REVIEW event is generated, and the system is in MODIFY_DESIGN. Static reactions extend the power of the states by allowing reactions to be defined for system events.

In Section 4 we show how the statecharts can be compiled into a MARVEL environment; first, we give a brief introduction to the MARVEL system.

3 Marvel approach

The goal of the MARVEL project [1] is to develop *process-centered environments* that guide and assist teams of users working on large-scale projects. An *administrator* provides the schema, process model, tool envelopes and coordination model for a specific project. The schema classes define an objectbase containing the system under development. Multiple inheritance, status attributes of primitive and enumerated types, file attributes of text and binary types, aggregate composite objects, and links between objects can be declared.

The process is specified in a *process modeling language*. Each process step is encapsulated in a *rule*, whose name and typed parameters are used to generate the command menu. Each rule is composed of a condition, an optional activity, and a set of effects. The condition has two parts: local *bindings* to query the objectbase to gather implicit parameters and a *property list* that must evaluate to `true` prior to invocation of the step. Consider a rule which compiles a C source code file by first gathering all the included `".h"` files and verifying that the source code file has first been statically analyzed by the `lint` tool. Each effect asserts one of the step's possible results, in this case, whether the file compiled successfully or not.

Forward and backward chaining over the rules enforces consistency in the objectbase and automates tool invocations. When the user invokes a rule r whose condition is not satisfied, *backward chaining* is initiated in an attempt to satisfy r's condition. Note that there is a specific predicate p which caused r's condition to fail. The rule engine collects together into a set S_b those rules that have an effect that satisfies p. The engine then repeatedly removes and invokes r_i from this set until either p is satisfied or S_b is empty. Since backward chaining is a recursive process, the engine might initiate a backward chain to satisfy the condition of r_i. This backward chaining process is repeated until the condition of the original rule r is satisfied, or all possibilities are exhausted.

Forward chaining is initiated when a predicate p in the effect of a rule is asserted on the objectbase. The engine determines those rules $r_i \in S_f$ whose condition have become satisfied because of this assertion. The rules are compiled into a rule network, so these rules r_i can be efficiently determined. Each of these rules is invoked, and, in a recursive fashion, other rules are added to S_f until all possible forward chains are executed. A rule network is compiled when the MARVEL system is tailored by the prcoess model. The process engine chains among rules with different or multiple parameters by "inverting" local bindings [6].

Conventional file-oriented tools are integrated through an *enveloping* language based on shell scripts [5]. A rule activity specifies an envelope and its input and output arguments, which may be literals, status attributes and/or (sets of) file attributes. Each envelope returns a distinguished value to indicate which of the specified effects should be asserted.

The MARVEL client provides the user interface, checks the arguments of commands, and forks operating system processes to execute envelopes. The process engine, synchronization, and object management system reside in the MARVEL server. Scheduling is first-come first-served, with rule chains interleaved at the natural breaks when clients execute rule activities. In particular, the process engine executes the chain initiated by a client's command until an activity is encountered. Sufficient information to execute the activity is then returned to that client, and the server retrieves the next client request. When an activity terminates, the client places its results in the server's queue, enabling the server to resume its in-progress chain.

For more detailed information regarding the MARVEL system, the interested reader should inspect the referenced papers. This brief introduction covers the details of MARVEL necessary for this paper. We now proceed to discuss how the statecharts are compiled into a MARVEL environment.

4 Compiling SEI into Marvel

The goal of the compilation is to produce a MARVEL environment that exhibits the same behavior as the statechart. Each state in the statechart is compiled into a STATE entity in the objectbase. State decomposition is represented through the *substates* composition attribute by allowing states to have children STATE objects. Each state has a link attribute, *active*, that links to its active substate. In the case of OR-states, there can only be one active substate, hence the OR_STATE overrides the *active* link to be a singleton set. In order to reproduce the behavior of statecharts, each STATE has several attributes that are used during chaining.

- *clear_deep_history* is `true` when the deep-history for the state is being cleared
- *clear_history* is `true` when the normal history for the state is being cleared.
- *set_history* is `true` when the history for a state is being saved.
- *activate_history* is `true` when the history for a state is being restored.
- *activate_defaults* is `true` when the state has substates, and a state transition has been made to it.
- *entering* is `true` when a transition has just been made to the state
- *exiting* is `true` when a transition has been made from the state
- *connectors* maintains the history, deep-history, and in some cases condition connectors for a state.

The mapping from a statechart to the objectbase is straightforward. STATE objects are created for each state, belonging to the appropriate subclass, and having the appropriate parent STATE object as determined by the state decomposition. History

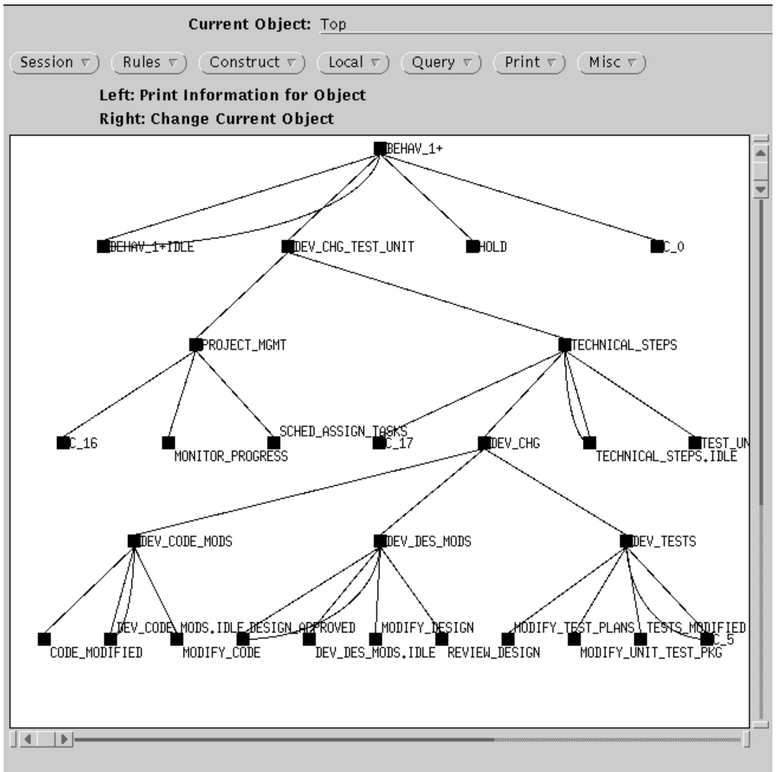

Figure 2: Marvel objectbase for statechart

connectors are created, belonging to the *connectors* composition attribute of the appropriate states. All default entrances are modeled by a link from the parent STATE object to the appropriate STATE through the *default_entrance* link object. For example, TECHNICAL_STEPS has a link to TECHNICAL_STEPS.IDLE. Condition connectors are only created when they are used as part of the default transition of a state (as in DEV_TEST). In this case, the state links to the condition connector through its *default_connector* attribute. Conditions (and data items) are each separately instantiated as objects belonging to the CONDITION (and DATA_ITEM) class. Figure 2 shows the objectbase resulting from compiling the statechart in figure 1.

4.1 Transitions

The central part of the compilation involves translating state transitions into MSL rules that alter the objectbase to reflect the changes in the system. Each transition has three parts (any of which may be empty): a triggering event, a restricting condition, and an action. STATEMATE allows each of these to be constructed using AND, OR and NOT expressions. These are each described further in the following subsections. We now describe the mapping from transitions to rules.

Figure 3 contains the MSL rule corresponding to the transition from IDLE to RE-

```
## from DEV_DES_MODS.IDLE to REVIEW_DESIGN "TIME_FOR_REVIEW"
1 hide rule_7[?e:EVENT_LIST]:
2   (and (exists GROUP ?Top suchthat no_chain (member [?Top.events ?e]))
3        (exists STATE ?REVIEW_DESIGN suchthat
4             (and no_chain (ancestor [?Top ?REVIEW_DESIGN])
5                  no_chain (?REVIEW_DESIGN.Name = "REVIEW_DESIGN")))
6        (exists STATE ?DEV_DES_MODS suchthat
7             (and no_chain (ancestor [?Top ?DEV_DES_MODS])
8                  no_chain (?DEV_DES_MODS.Name = "DEV_DES_MODS")))
9
10     # Currently in "DEV_DES_MODS.IDLE" state
11      (exists STATE ?DEV_DES_MODS_IDLE suchthat
12            (and no_chain (linkto [?DEV_DES_MODS.active ?DEV_DES_MODS_IDLE])
13                 no_chain (?DEV_DES_MODS_IDLE.Name = "DEV_DES_MODS.IDLE")))):
14
15  (and          (?e.TIME_FOR_REVIEW = true)
16       no_chain (?DEV_DES_MODS_IDLE.entering = false))
17
18 { }
19
20 (and no_chain (unlink [?DEV_DES_MODS active ?DEV_DES_MODS_IDLE])   # Switch states
21               (?DEV_DES_MODS_IDLE.exiting = true)
22      no_chain (link [?DEV_DES_MODS active ?REVIEW_DESIGN])
23               (?REVIEW_DESIGN.entering = true)                  # Mark as transitory
24               (?REVIEW_DESIGN.activate_defaults = true));
```

Figure 3: MSL rule for transition from SCHED_ASSIGN_TASKS to MONITOR_PROGRESS

VIEW_DESIGN. The MARVEL rule representing this transition, `rule_t` has four parts: the binding-phase, property list, activity, and effects. The binding-phase (lines [2-13]) queries the objectbase for information needed by the rule. In `rule_7`, the system must be in IDLE[2] for the transition to take place. This is handled in [11-13] by binding the variable `?DEV_DES_MODS_IDLE` to the active child of `?DEV_DES_MODS` whose name is `DEV_DES_MODS.IDLE`. If `DEV_DES_MODS` links to `IDLE` through its *active* link, then `?DEV_DES_MODS_IDLE` is bound to the object `IDLE`, otherwise the binding-phase fails and the rule is not applicable. If, however, this link does exist, the binding-phase succeeds, and the property list is evaluated to see if the TIME_FOR_REVIEW event has been generated. There is also a check to make sure that DEV_DES_MODS_IDLE has not just been entered, since that would mean the event has already caused a transition (into DEV_DES_MODS_IDLE) and should not mistakenly be acted on twice. Once the property list [15-16] evaluates to `true`, the rule proceeds with the effects [20-24] since there is no activity [18]. The old link to DEV_DES_MODS_IDLE is removed, and REVIEW_DESIGN is made the active substate by setting the link. Assertions are made to register the fact that DEV_DES_MODS_IDLE is being exited, while REVIEW_DESIGN is being entered. One final assertion is made to activate the substates, if any, of the state being entered.

`rule_7` is the simplest example of a compiled transition. A more complicated transition is the one from MONITOR_PROGRESS to HOLD. Figure 4 contains the MSL rule `rule_3` corresponding to this transition. The binding-phase is considerably more

[2]To uniquely identify the IDLE state its name is extended to be "parent.state"

```
## from MONITOR_PROGRESS to HOLD "RECOMMEND_CANCEL/wr!(REC_CANCEL)"
hide rule_3[?e:EVENT_LIST]:
(and (exists GROUP ?Top suchthat no_chain (member [?Top.events ?e]))
     (exists STATE ?HOLD suchthat
           (and no_chain (ancestor [?Top ?HOLD])
                no_chain (?HOLD.Name = "HOLD")))
     (exists STATE ?BEHAV_1_ suchthat
           (and no_chain (ancestor [?Top ?BEHAV_1_])
                no_chain (?BEHAV_1_.Name = "BEHAV_1+")))
     (exists STATE ?PROJECT_MGMT suchthat
           (and no_chain (ancestor [?Top ?PROJECT_MGMT])
                no_chain (?PROJECT_MGMT.Name = "PROJECT_MGMT")))
     (exists CONNECTOR ?C_16 suchthat no_chain (?C_16.Name = "C_16"))

     # Currently in "MONITOR_PROGRESS" state
     (exists STATE ?MONITOR_PROGRESS suchthat
           (and no_chain (linkto [?PROJECT_MGMT.active ?MONITOR_PROGRESS])
                no_chain (?MONITOR_PROGRESS.Name = "MONITOR_PROGRESS")))
     (exists INTEGER_DATA ?REC_CANCEL suchthat
           (and no_chain (ancestor [?Top ?REC_CANCEL])
                no_chain (?REC_CANCEL.Name = "REC_CANCEL")))
     (exists STATE ?DEV_CHG_TEST_UNIT suchthat
           (and no_chain (ancestor [?Top ?DEV_CHG_TEST_UNIT])
                no_chain (?DEV_CHG_TEST_UNIT.Name = "DEV_CHG_TEST_UNIT")))
     (exists STATE ?TECHNICAL_STEPS suchthat
           (and no_chain (ancestor [?Top ?TECHNICAL_STEPS])
                no_chain (?TECHNICAL_STEPS.Name = "TECHNICAL_STEPS")))
     (exists CONNECTOR ?C_17 suchthat no_chain (?C_17.Name = "C_17"))

     # Get active links for PROJECT_MGMT
     (exists STATE ?PROJECT_MGMT_L_I_N_K_S suchthat
           no_chain (linkto [?PROJECT_MGMT.active ?PROJECT_MGMT_L_I_N_K_S]))
     # Get active links for TECHNICAL_STEPS
     (exists STATE ?TECHNICAL_STEPS_L_I_N_K_S suchthat
           no_chain (linkto [?TECHNICAL_STEPS.active ?TECHNICAL_STEPS_L_I_N_K_S]))):

 (and          (?e.RECOMMEND_CANCEL = true)
     no_chain (?MONITOR_PROGRESS.entering = false))

 { }
 (and no_chain (unlink [?PROJECT_MGMT active ?MONITOR_PROGRESS])    # Switch states
               (?MONITOR_PROGRESS.exiting = true)
               # Need to save link in history object: C_16
               # Forward chaining is necessary since C_16 is DEEP
               (link [?C_16 states ?PROJECT_MGMT_L_I_N_K_S])
               # Need to save link in history object: C_17
               # Forward chaining is necessary since C_17 is DEEP
               (link [?C_17 states ?TECHNICAL_STEPS_L_I_N_K_S])
               # Need to make unlink visible since DEV_CHG_TEST_UNIT is an AND_STATE
               (unlink [?DEV_CHG_TEST_UNIT active ?PROJECT_MGMT])   # Switch states
               (?PROJECT_MGMT.exiting = true)
     no_chain (unlink [?BEHAV_1_ active ?DEV_CHG_TEST_UNIT])       # Switch states
               (?DEV_CHG_TEST_UNIT.exiting = true)
     no_chain (link [?BEHAV_1_ active ?HOLD])
               (?HOLD.entering = true)                              # Mark as transitory
               (?HOLD.activate_defaults = true)
               (?REC_CANCEL.written = true)                         # wr!(REC_CANCEL));
```

Figure 4: MSL rule for transition from IDLE to SCHED_ASSIGN_TASKS

```
+-------------------------------------------------------------------------+
| Transcript for: Interface for command: generate_AUTHORIZATION event(1)   |
|                                                                          |
| Attempting: generate_AUTHORIZATION(event)                                |
|     Forward: rule_1(event)                                               |
|       Forward: activate_default_2(DEV_CHG_TEST_UNIT)                     |
|         Forward: activate_default_1(TECHNICAL_STEPS)                     |
|           Forward: initialize_history(C_17)                              |
|             Forward: initialize_history(C_16)                            |
|               Forward: clear_AUTHORIZATION(event)                        |
|                 Forward: reset_states(event)                             |
+-------------------------------------------------------------------------+
```

Figure 5: Rule chain resulting from initial transition

complex (lines [2-34]) since the transition is a more complex one. This transition must deactivate the DEV_CHG_TEST_UNIT state which affects the history objects (in both PROJECT_MGMT and TECHNICAL_STEPS); care must be taken to save all necessary state information so it can be restored when a transition is made back to DEV_CHG_TEST_UNIT through the CCB_RESUME event. Lines [15-17] enforce this transition to occur only when MONITOR_PROGRESS is the active state. History information is retrieved by lines [30-34] and saved in lines [44, 47]. The necessary exiting [41, 50, 52] and entering [54] events are asserted, and the HOLD state is made active [51, 53]. Both the MONITOR_PROGRESS [40] and PROJECT_MGMT [49] states are made inactive. Finally, the wr!(REC_CANCEL) action is carried out [56] and any substates of HOLD are activated.

Each transition is compiled into one rule, but there are other helper rules that make sure that the state of the system is accurately represented. The transition rules are found in the `chart.load` strategy file, while the helper rules are found in `statemate.load`. These helper rules are common across all state chart compilations. To give examples of how these rules work, we now show the rule chains that result from particular scenarios in the statechart. In the next section we show the helper rules that correctly activate states when transitions are made. Since transitions are concerned with events, conditions, and actions, we then describe each of these in turn and show the rule chains that occur to produce the necessary behavior.

4.2 State decomposition

In the compiled MARVEL environment, each state maintains information about its active substates. AND-states can have many active substates; OR-states can only have one. Each state, therefore, has a link attribute, *active*, that links a state to its active substate(s). When transitions are made between states, these links need to be appropriately maintained since they determine the state of the process. Six helper rules maintain the links by detecting when transitions are made into and out of states.

`activate_default_1`	When a transition is made to the outer edge of a state which is further decomposed, the substate containing the default entrance is entered.
`activate_default_2`	When an AND-state is entered, all of its concurrent substates are entered.
`activate_default_3`	A transition is made to a primitive state with no substates.
`activate_default_4`	When the default entrance of an OR-state is to a condition connector, the connector is activated to select the appropriate branch.
`activate_default_5`	Terminates recursion process.
`deactivate_substates`	When a transition is made from the outer edge of a state which is further decomposed, all substates are deactivated.

Table 1: Helper rules for transitions involving substates

The basic principles, and the helper rules that realize them, are shown in Table 1. Consider the initial transition in the statechart, from IDLE to SCHED_ASSIGN_TASKS as shown in Figure 5. This is an implicit transition to the DEV_CHG_TEST_UNIT state, therefore `activate_default_2` is fired since the state is an AND-state. This causes TECHNICAL_STEPS to be entered, hence `activate_default_1` is fired. The other rules regarding the history connectors and events are described more fully in Section 4.6 and 4.3.

4.3 Events

Events are either simple or complex. Simple events can be treated as boolean values that are set to `true` when the event is raised, and reset to `false` when the event falls. The AUTHORIZATION event belongs to this category. Complex events are the following:

wr(D)	The data item D has been written
rd(D)	The data item D has been read
ch(D)	The value of data item D has been changed
en(S)	The system is now entering state S
ex(S)	The system is now exiting state S
E[C]	Event E has been generated, and condition C is `true`

```
EVENT_LIST :: superclass ENTITY;
  AUTHORIZATION    : boolean = false;
  CCB_RESUME       : boolean = false;
  CCB_CANCEL       : boolean = false;
  RECOMMEND_CANCEL : boolean = false;
end
```

Figure 6: The generated class EVENT_LIST

```
+-------------------------------------------------------------------------+
|   generate_TIME_FOR_REVIEW[?e:EVENT_LIST]:                              |
|     :                                                                   |
|     { }                                                                 |
|     (and no_chain (?e.TIME_FOR_REVIEW = false)                          |
|                   (?e.TIME_FOR_REVIEW = true));                         |
|                                                                         |
|   -------------------------------------------------------------------- |
|   Transcript for: Interface for command: generate_TIME_FOR_REVIEW event(1) |
|                                                                         |
|   Attempting: generate_TIME_FOR_REVIEW(event)                           |
|       Forward: rule_7(event)                                            |
|         Forward: clear_TIME_FOR_REVIEW(event)                           |
|           Forward: reset_states(event)                                  |
+-------------------------------------------------------------------------+
```

Figure 7: Rule chain for transition from IDLE to REVIEW_DESIGN

The condition "E1 or E3[C] and en(S) or ex(S) and en(T)" is equivalent to:

> Event E1 has been raised; or event E3 has been raised and condition C is `true` and the system is entering state S; or the system is exiting state S and entering state T.

Events are processed in serial order – this implies that it is impossible to generate two events simultaneously; hence events such as "E1 and E2" make no sense.

The MARVEL environment generated by the compiler has a class EVENT_LIST that contains attributes for each simple event; for the ISPW-6 example, the EVENT_LIST class is shown in figure 6. STATEMATE allows for such events as "not CCB_RESUME and not CCB_CANCEL"; this particular example could be compiled into "AUTHORIZATION or RECOMMEND_CANCEL" using simple logic, but the compiler does not currently support this. The objectbase has one object `event` belonging to the EVENT_LIST class, thus reflecting the belief that events cannot occur simultaneously.

The rule chain produced by generating the TIME_FOR_REVIEW event is shown in figure 7. Once the appropriate event has been generated (`generate_TIME_FOR_REVIEW`), the system detects a forward chain to `rule_7` (on the predicate `(?e.TIME_FOR_REVIEW = true)`) and executes the rule. Once all applicable rules have been fired two helper rules which reset the state of the objectbase for the next event. `reset_states` resets the values of the *entering* and *exiting* attributes of those states which were affected by any transitions during the rule chain and `clear_TIME_FOR_REVIEW` resets the TIME_FOR_REVIEW event.

4.4 Conditions

Conditions are either simple or complex. Simple conditions inspect the value of booleans. The "[REWORK_TEST]" condition belongs to this category. This condition is `true` if the REWORK_TEST value is `true`. Complex conditions are logical combinations of simple conditions and mathematical expressions over the set of data_items for the state chart. Sometimes a condition is described as a compound condition, for example, APPROVED is described as the following compound:

```
PASSED and COVERAGE_ATTAINED>.9
```

Compound conditions are separated into their individual parts when compiled into rules. A transition with a condition but no event is triggered when the state is entered, and the condition is `true`. Consider the event from DEV_CODE_MODS.IDLE to MODIFY_CODE. If the condition [CLEAN and APPROVED and CODE_REFLECTS_DES] were `true`, the system would make a transition from MODIFY_CODE to CODE_MODIFIED. As an additional example, the default transition from DEV_TESTS goes to MODIFY_UNIT_TEST_PKG when [REWORK_TESTS] is `true`, and to TEST_MODIFIED when [not REWORK_TESTS] is `true`.

The MARVEL environment generated by the compiler creates a separate object for each condition, belonging to the CONDITION class. There are no helper rules for conditions since the condition will remain `true` (or `false`) until some other rule changes it.

4.5 Actions

Actions are performed once the transition has been made. The following actions are compiled into the MARVEL environment:

dc!(H)	Clear the deep history connector of its state information
hc!(H)	Clear the history connector of its state information
wr!(D)	write data item D
rd!(D)	read data item D
tr!(C)	Make condition C `true`
fs!(C)	Make condition C `false`

```
+-------------------------------------------------------------------+
| Transcript for: Interface for command: write_data_item MOD_DESIGN(36) |
|                                                                   |
| Attempting: write_data_item(MOD_DESIGN)                           |
|    Forward: rule_6(event)                                         |
|      Forward: clear_written(MOD_DESIGN)                           |
+-------------------------------------------------------------------+
```

Figure 8: Rule chain for wr!(data item)

```
+------------------------------------------------------------------------+
| Transcript for: Interface for command: generate_RECOMMEND_CANCEL event(1)  |
|                                                                        |
| Attempting: generate_RECOMMEND_CANCEL(event)                           |
|     Forward: rule_3(event)                                             |
|       Forward: set_history1(DEV_CHG)                                   |
|         Forward: set_history1(DEV_TESTS)                               |
|           Forward: set_history1(DEV_DES_MODS)                          |
|             Forward: set_history1(DEV_CODE_MODS)                       |
|               Forward: deactivate_substates(BEHAV_1+)                  |
|                 Forward: clear_written(REC_CANCEL)                     |
|                   Forward: clear_RECOMMEND_CANCEL(event)               |
|                     Forward: reset_states(event)                       |
+------------------------------------------------------------------------+
```

Figure 9: Example use of history connectors

Actions do not require separate entities in the MARVEL objectbase. STATEMATE allows actions to generate events, but for simplicity, the compiler ignores them. Figure 8 shows the rule chain invoked when a data item is written. `clear_written` is a helper rule that restores the data item to its original (i.e., unread, unwritten) state.

4.6 History Connectors

Statecharts provide a memory mechanism called History Connectors to reinstate a once exited configuration. The principles governing their behavior is as follows:

- A state, when exited and reentered through a history connector, will return to the substate on the same level as the history connector which last had control.
- A state, when exited and reentered through a deep history connector, will return to all the states at all levels which last had control.
- Clearing a history's memory is accomplished through the `hc!(state)` action. Clearing a deep history's memory is accomplished through the `dc!(state)` action.

```
CONNECTOR
  | states      : set_of link STATE;
  | initialized : boolean = false;
  | activate    : boolean = false;
  |
  +--TERMINATION_CONNECTOR
  |
  +--CONDITION_CONNECTOR
  |     active : boolean = false;
  |
  +--HISTORY_CONNECTOR
  |     clearing : boolean = false;
  |
  +--DEEP_HISTORY_CONNECTOR
        clearing : boolean = false;
```

Figure 10: Class inheritance of the connectors

`set_history_1`	When a deep history connector is activated, save all of the substates in the deep history connector's *states* attribute.
`set_history_3`	When a history connector is activated, save the substates in the history connector's *state* attribute.
`initialize_history`	The first time a history is encountered, initialize it.
`activate_history_1`	When a history connector is activated, mark all active states maintained in the history.
`activate_history_3`	Sets the parent *active* link to link to the state which has just been marked, and which was just in a deep history's *state* list.
`activate_history_4`	Sets the parent *active* link to link to the state which has just been marked, and which was just in a history's *state* list. The difference with the previous rule is that this rule can spawn chains to activate default entrances.
`finish_history`	Once all states have been restored, this rule removes the links from the history list.
`take_default`	When a transition is made to an uninitialized history the default transition occurs.
`take_default2`	When a transition is made to an uninitialized history and the default transition is to a condition connector, then activate the connector.

Table 2: Helper rules for transitions involving history connectors

```
+-----------------------------------------------------------------------+
| Transcript for: Interface for command: generate_CCB_RESUME event(1)   |
|                                                                       |
| Attempting: generate_CCB_RESUME(event)                                |
|     Forward: fanout_2(event)                                          |
|       Forward: activate_history_1(C_17)                               |
|         Forward: activate_history_3(DEV_TESTS)                        |
|           Forward: activate_history_3(DEV_DES_MODS)                   |
|             Forward: activate_history_3(DEV_CODE_MODS)                |
|               Forward: activate_history_3(DEV_CHG)                    |
|                 Forward: activate_history_3(MODIFY_TEST_PLANS)        |
|                   Forward: activate_history_3(MODIFY_DESIGN)          |
|                     Forward: activate_history_3(DEV_CODE_MODS.IDLE)   |
|                       Forward: finish_history(C_17)                   |
|                         Forward: activate_history_1(C_16)             |
|                           Forward: activate_history_3(MONITOR_PROGRESS)  |
|                             Forward: finish_history(C_16)             |
|                               Forward: clear_CCB_RESUME(event)        |
|                                 Forward: reset_states(event)          |
|                                                                       |
+-----------------------------------------------------------------------+
```

Figure 11: Example use of restoring state information from history

In order to simplify the process of compiling transitions into rules, helper rules are provided which maintain the history information. Each history connector is instantiated in the objectbase as a child of the state to which it is attached. In our example, the deep history connector of PROJECT_MGMT appears as object `c_16`. Each CONNECTOR object has a link attribute, *states*, which links to the states it is maintaining. The HISTORY_CONNECTOR and DEEP_HISTORY_CONNECTOR are subclasses of CONNECTOR. The *initialized* attribute is `true` when the history (or deep history) connector has been initialized. This is important to remember since the transition to an uninitialized history connector forces the parent state to make a default transition. The *clearing* attribute is `true` when the history information is being cleared.

The rule chain in Figure 9 shows how these helper rules operate. The assertion in line [47] of `rule_3` causes a forward chain to `set_history_1` which recursively is fired three additional times to save all the state information with the deep history connector `c_17`. The rule chain in Figure 11 shows how the history is restored when a transition is made to a history connector. The assertions from the `fanout_2` rule (not shown) activate the history objects `c_17` and `c_16`. `activate_history_3` restores the active links for all necessary substates of TECHNICAL_STEPS and `finish_history` clears the history. In similar fashion, the history for PROJECT_MGMT is restored.

4.7 Condition Connectors

Condition connectors allow a transition to be split based upon differing conditions. Transitions to condition connectors are not individually compiled into rules; rather,

```
+--------------------------------------------------------------------+
| Transcript for: Interface for command: generate_CCB_CANCEL event(1)  |
|                                                                    |
| Attempting: generate_CCB_CANCEL(event)                             |
|     Forward: terminate_1(event)                                    |
|       Forward: statechart_termination(C_0)                         |
|         Forward: history_termination(C_17)                         |
|           Forward: history_termination(C_16)                       |
|             Forward: reset_termination(C_0)                        |
|               Forward: clear_CCB_CANCEL(event)                     |
|                 Forward: reset_states(event)                       |
+--------------------------------------------------------------------+
```

Figure 12: Example use of termination connectors

`statechart_termination`	When a termination connector is activated, force all history objects to become initialized.
`history_termination`	Clear all state information for a history connector.
`reset_termination`	Deactivate a termination connector once all states in the statechart are inactive.

Table 3: Helper rules for transitions involving termination connectors

transitions are made to each possible target of the connector (potentially a recursive process). Consider the transition from TEST_UNIT when the wr(TEST_RESULTS) event is generated. There are three possible targets which can be reached: CODE_MODIFIED, MODIFY_CODE, and the termination connector (labeled "T"). Therefore, three rules are output whose conditions are merged to contain all conditions along each path:

CODE_MODIFIED	wr(TEST_RESULTS)[not SUCCESS and not REWORK_CODE]
MODIFY_CODE	wr(TEST_RESULTS)[not SUCCESS and REWORK_CODE]
`c_0` (Termination)	wr(TEST_RESULTS)[SUCCESS]

No transition is made if none of these logic expressions evaluates to `true`.

When the default entrance for a state is a condition connector, as in DEV_TEST, some small complications arise. In these situations, the condition connector is instantiated as a child object to the specified state. Here, the `activate_default_4` rule from Figure 1 is fired when a transition is made to DEV_TESTS and the condition connector `c_5` is activated. A rule is generated for each of the transitions out of the condition connector (one of which must be satisfied), and the appropriate one is fired when the condition connector is activated.

4.8 Termination Connectors

Termination connectors complete the action for a statechart and restores it to its uninitialized state. When a transition to a termination connector occurs, the connector is activated, which causes helper rules to fire. The rule chain in Figure 12 shows the sequence of rule firings which restore the statechart to its uninitialized state. To reactivate the statechart, the user must execute the `instantiate` rule on the STATECHART object.

4.9 Static Reactions

Static reactions are associated with states and are handled when the state is active. There are two forms of reactions, those which are triggered when the state is entered or exited, and those which are triggered by an event occurring while the state is active.

- entering/S
- exiting/S
- E[C]/S

Here, S describes an arbitrary expression allowed by the STATEMATE syntax. This includes such expressions as:

```
if DES_REV_OUTCOME>1 then wr!(DES_REV_FDBK) end if
DES_REV_OUTCOME:=0
if PASSED and COVERAGE_ATTAINED>=.9 then wr!(TEST_SUCCESS) end if
if N_REV=1 then sc!(DES_REV_OUTCOME:=2;wr!(DES_REV_RESULTS,.375) end if
ch(RND)[N_REV=0]/if RND<.1 then NEW:=1 else if RND<.4 then NEW:=2 else NEW:=3 end if end if
```

Such static reactions, and in particular, all static reactions, are best delegated to exist in the SEL scripts of appropriate rules. As a simulation tool, STATEMATE needs a way of generating appropriate events and scheduling them at future times – this is done using the "sc!(x)" action. Such an action is not necessary in the MARVEL environment, since that event will be generated once the actual activity has occurred. For example the following static reaction produces the "wr!(DES_REV_RESULTS)" action 3/8ths of a time unit in the future if the value of N_REV is equal to 1.

```
if N_REV=1 then sc!(DES_REV_OUTCOME:=2;wr!(DES_REV_RESULTS,.375) end if
```

This simulates the situation which arises when the number of revisions needed is minor (that is, one), and the time to finish these revisions will only be 3/8ths of an hour. In practice, the revisions will be initiated, and the event will be generated upon completion of the revisions. These static reactions, however, are not wasted; they can be used by the manager to inspect the process to check actual results against anticipated expectations.

5 Future work

The compiler we have just described takes three STATEMATE descriptions – Behavioral, Functional, and Organizational – and produces a MARVEL environment which exhibits the same behavior. This proved to be a valuable experience in translating from one modeling formalism to another. Extensions to this work are many, and include the following:

- *Incorporate Static Reactions* – In Kellner's work [9], static reactions produce quantitative measurements for simulation purposes. These could be stored and correlated with the actual process as it is being enacted.
- *Incorporate Organizational Perspective* – The MARVEL environment does not allows rule chains *between* users (i.e., delegation of tasks). We made some preliminary investigations on the possibility of using Process WEAVER [4] which suggest that it could be used to perform the necessary delegation.
- *Incorporating Tools into the Process Model* – The generated environment uses no external tools. To fully realize modeling technology, a process engineer must be able to model not only the process flow, but the tools to be used. MARVEL provides an excellent facility for tool support which could be utilized in this fashion.

Acknowledgments

We would like to thank Marc Kellner and David Raffo for providing the STATEMATE statecharts used in this paper. We would also like to thank the IBM Centre for Advanced Studies (CAS) in Toronto, and in particular John Botsford, for providing an environment without which this research would not have been realized.

References

[1] Israel Z. Ben-Shaul, Gail E. Kaiser, and George T. Heineman. An architecture for multi-user software development environments. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 149–158, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[2] David Harel et al. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[3] Peter H. Feiler and Gail E. Kaiser. Intelligent Assistance in Software Development Environments. In *Annual Technical Review 1987*, pages 43–56, Pittsburgh, PA, 1987. Software Engineering Institute, Carnegie Mellon University.

[4] Christer Fernstrom. Process weaver: Adding process support to unix. In *1st International Conference on the Software Process: Continuous Software Process Improvement*, pages 12–26, Berlin, Germany, February 1993. IEEE Computer Society Press.

[5] Mark A. Gisi and Gail E. Kaiser. Extending a tool integration language. In Mark Dowson, editor, *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 218–227, Redondo Beach CA, October 1991. IEEE Computer Society Press.

[6] George T. Heineman, Gail E. Kaiser, Naser S. Barghouti, and Israel Z. Ben-Shaul. Rule chaining in marvel: Dynamic binding of parameters. *IEEE Expert*, 7(6):26–32, December 1992.

[7] Marc I. Kellner. Software process modeling: Value and experience. In *In SEI Technical Review, Software Engineering Institute*, pages 23–54. Carnegie Mellong University, 1989.

[8] Marc I. Kellner. Multiple - paradigm approaches for software process modeling. In *7th International Software process Workshop: Communication and Coordination in the Software Process*, pages ??–??, Yountville, CA, October 1991. IEEE Computer Society.

[9] Marc I. Kellner. Software process modeling support for management planning and control. In *1st International Conference on the Software Process: Manufacturing Complex Systems*, Redondo Beach CA, August 1991.