

# Automatic Translation of Process Modeling Formalisms

George T. Heineman  
Columbia University

## Abstract

This paper<sup>1</sup> demonstrates that the enactment of a software process can be separated from the formalism in which the process is modeled. To show this, we built a compiler capable of automatically translating a process model specified using Statemate into a MARVEL environment which then supports the enactment of the process.

## 1 Introduction

The software community has recognized the importance of understanding and modeling the processes used to develop and support software. There are many objectives that have been cited as motivation for the development and application of software process models including support for automated execution and control, human interaction, managerial responsibilities, process understanding, and analysis of processes [7]. There is a problem, however, in deciding which process modeling formalism to use as there are dozens available. This paper proposes a possible solution by suggesting that the actual enactment 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 that must be manually carried out by individual users, but in many cases,

---

<sup>1</sup>The IBM contact for this paper is John Botsford, Centre for Advanced Studies, IBM Canada Ltd., Dept. 21/894, 844 Don Mills Road, North York, Ontario M3C 1V7.

there is no way to monitor or control the processes, and the managers must simply hope that all the users follow the specified processes.

We present a compiler that translates a process modeled using the SEI-developed Statemate<sup>2</sup>-modeling approach into an active MARVEL environment that can be used to enact the process. Section 2 of this paper describes the SEI software process modeling approach and partially reproduces Kellner's solution to the ISPW-6 standard benchmark problem [8]. Section 3 describes the MARVEL approach to software modeling. Section 4 describes the compiler which translates the Statemate specification of a process into a MARVEL environment. Section 5 outlines some future research directions.

## 2 SEI Statemate approach

The SEI Statemate-approach [6] models software processes by constructing three interrelated perspectives that determine the who, what, where, when, and how of a process:

- Functional – what the tasks are, and the information flow between the tasks
- Behavioral – when and how tasks are performed
- Organizational – 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

---

<sup>2</sup>Statemate is a trademark of i-Logix, Inc., Burlington, MA.

focus on the activities being performed and hide the actual details of how they are carried out. Statecharts<sup>3</sup> provide the behavioral perspective and focus on the timing and ordering of individual process steps. The final perspective is provided by module charts that 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. For example, the activity and module charts are connected to assign responsibilities for each activity. Also, sequencing of activities is determined by the connections between the activity charts and statecharts. Since this paper focuses on the compilation of statecharts, we now describe them in more detail.

Statecharts [2] 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 sub-states. If the subdivision represents a concurrent decomposition 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 in a substate of each of its orthogonal subcomponents. Each transition can have a condition associated with it that 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 (fan-in), restricting them to occur only when the system is in each of the source states, or multiple targets (fan-out), which activates a set of target states when the transition is made. Finally, statecharts can have history connectors which maintain state information, simplifying the writing of interrupts and special events.

Stepping through the statechart in Figure 1 will clearly explain how statecharts

work. This 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. A transition into the history connector causes its owner 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]. The condition value directs transitions to different targets.

---

<sup>3</sup>Statemate uses this term instead of state chart.

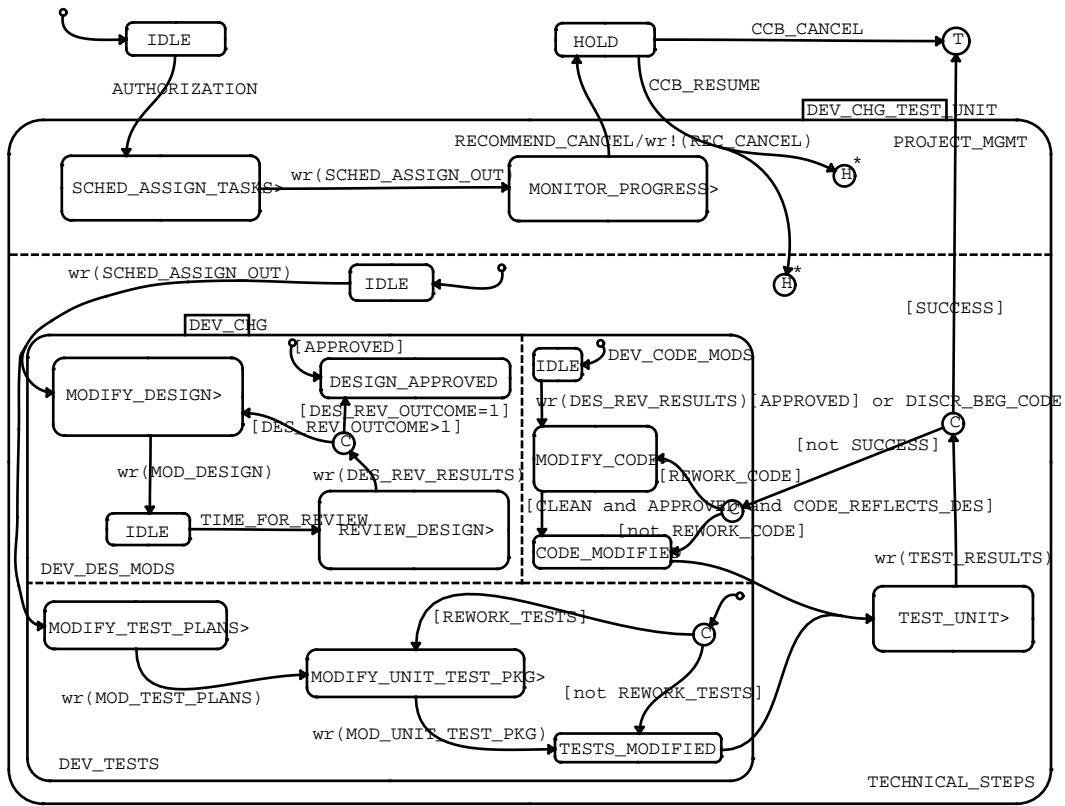


Figure 1: Sample Statechart

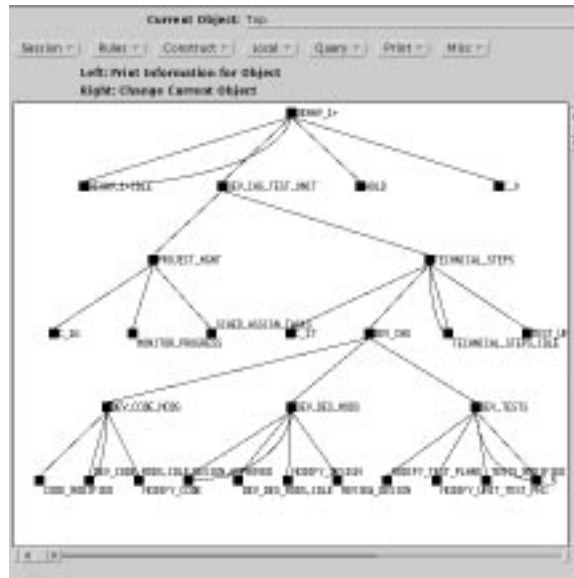


Figure 2: MARVEL objectbase for statechart in Figure 1

Statecharts can also have static-reactions associated with each state that can observe and influence the system; they are not transitions, since 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 to react to system events. In Section 4 we show how 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 a Process Centered Environment that guides and assists 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 in the schema.

The schema and process are specified in MARVEL's process modeling language, MSL (MARVEL strategy language). Each process step is encapsulated in a *rule* with typed parameters. 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 to compile a C source code file that gathers all its `#include ".h"` files; the rule might have a condition specifying that the C file must successfully pass a static code analyzer (such as the `lint` tool). Each effect of a rule 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. MARVEL's rule engine collects together those rules that have an effect that satisfies *p*. The engine then repeatedly removes and invokes rules from this set until either *p* is satisfied or the set is empty. Backward chaining is recursive and is executed until the condition of the original rule *r* is satisfied or all possibilities are exhausted.

*Forward chaining* is initiated when an effect of a rule is asserted on the objectbase. The engine determines the set of rules whose condition becomes satisfied because of this assertion. Since the rules are compiled in a rule network, this set can be efficiently determined. Each of the rules in this set is invoked, and, in recursive fashion, other rules are added to the set until all possible forward chains are exhausted. The rule engine chains among rules with different or multiple parameters by "inverting" local bindings [5].

Conventional file-oriented tools are integrated through an *enveloping* language based on shell scripts [4]. 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 code indicating which of the rule's effect to assert.

MARVEL is a client-server system. The client provides the user interface, checks the arguments of commands, and forks operating system processes to execute envelopes. The rule engine, synchronization, and object management system reside in the server. Scheduling

```

GROUP :: superclass ENTITY;
  charts      : set_of STATECHART;
  events      : EVENT_LIST;
end

### This class generated for the BEHAV_1 statechart
EVENT_LIST :: superclass STM_EVENT;
  CCB_CANCEL      : boolean = false;
  CCB_RESUME      : boolean = false;
  AUTHORIZATION   : boolean = false;
  RECOMMEND_CANCEL : boolean = false;
  TIME_FOR_REVIEW : boolean = false;
  DISCR_BEG_CODE  : boolean = false;
end

STATECHART :: superclass ENTITY;
  states      : STATE;
  data_items  : DATA_ITEM_LIST;
  data_entity : DATA_ENTITY;
  conditions  : CONDITION_LIST;
end

STATE :: superclass ENTITY;
  substates   : set_of STATE;
  active      : set_of link STATE;
  default_entrance : set_of link STATE;
  default_connector : link CONDITION_CONNECTOR;
  clear_deep_history : boolean = false;
  clear_history : boolean = false;
  set_history  : boolean = false;
  activate_history : (working, clearing, done) = done;
  activate_defaults : boolean = false;
  type        : (AND, OR, unknown) = unknown;
  entering    : boolean = false;
  exiting     : boolean = false;
  connectors  : set_of CONNECTOR;
end

```

Figure 3: Partial MSL schema

is first-come first-served, with rule chains interleaved at the natural breaks when clients execute rule activities. In particular, the rule 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. This brief introduction covers the details of MARVEL necessary for this paper. We now discuss how statecharts are compiled into a MARVEL environment.

## 4 Compiling statecharts

The goal of the compiler is to translate a given set of Statemate charts (State, Activity and Module) into “equivalent” MSL rules. By this we mean that the resulting MARVEL environment exhibits the same behavior as the statechart. Figure 3 contains a partial MSL schema generated by the compiler. Each state in the statechart is compiled into a STATE entity in the MARVEL 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 determines its active substate(s). 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 MARVEL objectbase is straightforward; Figure 2 shows the objectbase resulting from compiling the statechart in figure 1. 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 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.

```

# Generated by Compiler
# IDLE to REVIEW_DESIGN: "TIME_FOR_REVIEW"
1 hide rule_7[?e:EVENT_LIST]:
  (and
2   (exists GROUP ?Top suchthat
      no_chain (member [?Top.events ?e]))
3   (exists STATE ?R_D suchthat
4     (and no_chain (ancestor [?Top ?R_D])
5       no_chain (?R_D.Name = "REVIEW_DESIGN")))
6   (exists STATE ?DDM suchthat
7     (and no_chain (ancestor [?Top ?DDM])
8       no_chain (?DDM.Name = "DEV_DES_MODS")))

9 # Currently in "DEV_DES_MODS.IDLE" state
10 (exists STATE ?DDMI suchthat
11   (and no_chain (linkto [?DDM.active ?DDMI])
12     no_chain (?DDMI.Name = "IDLE")))
13 ):
14 (and (?e.TIME_FOR_REVIEW = true)
15   no_chain (?DDMI.entering = false))
16 { }
17 (and no_chain (unlink [?DDM.active ?DDMI])
18   (?DDMI.exiting = true)
19   no_chain (link [?DDM.active ?R_D])
20   (?R_D.entering = true)
   (?R_D.activate_defaults = true));

```

Figure 4: MSL rule for transition from SCHED\_ASSIGN\_TASKS to MONITOR\_PROGRESS

## 4.1 Transitions

The central part of statechart 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 4 contains the MSL rule, `rule_7`, corresponding to the transition from `IDLE` to `REVIEW_DESIGN`. `rule_7` has four parts: the binding-phase, property list, activity, and effects. The binding-phase (lines [2-12]) queries the objectbase for information needed by the rule. In `rule_7`, the system must be in `IDLE`<sup>4</sup> for the transition to take place. This is handled in [10-12] by binding the variable `?DDMI` to the active child of `?DDM` whose name is `DEV_DES_MODS.IDLE`. If `DEV_DES_MODS` links to `IDLE` through its *active* link, then `?DDMI` is bound to the object `IDLE`, otherwise the

<sup>4</sup>To uniquely identify the `IDLE` state, `rule_7` really refers to this state as `"DEV_DES_MODS.IDLE"`. For space reasons here, we shorten the name.

binding-phase fails and the rule is not applicable. If, however, this link does exist, the binding-phase succeeds. At this point, for example, `?R_D` is "bound" to the `REVIEW_DESIGN` state. The property list in [13-14] is then 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 evaluates to `true`, the rule proceeds with the effects [16-20] since there is no activity [15]. 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; more complicated transitions result in MSL rules of 60 or more lines. Each transition (arrow) is compiled into one rule, but there are other helper rules that make sure that the state of the system is accurately represented. The full MSL definition produced by the compiler is composed of standard class definitions, helper rules, and generated transition rules. These helper rules are common across all statechart 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 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,

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.

Figure 5: Helper rules for transitions involving substates

```

generate_AUTHORIZATION(event)
  Forward: rule_1(event)
    Forward: activate_default_2(DEV_CHG_TEST_UNIT)
      Forward: activate_default_1(TECHNICALSTEPS)
        Forward: initialize_history(C_17)
          Forward: initialize_history(C_16)
            Forward: clear_AUTHORIZATION(event)
              Forward: reset_states(event)

```

Figure 6: Rule chain resulting from initial transition

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.

The basic principles, and the helper rules that realize them, are shown in Figure 5. Consider the rule chain corresponding to the initial transition in the statechart, from IDLE to SCHED\_ASSIGN\_TASKS as shown in Figure 6. There 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 `TECHNICALSTEPS` to be entered, hence `activate_default_1` is fired.

```

# When an AND_state is marked as active, all of its
# subcomponents must enter their respective default states.
# This is a recursive process. NOTE: Only Substates which
# don't have active links already set are affected.
hide activate_default_2[?active:STATE];
  (forall STATE ?sub suchthat
    (and no_chain (member [?active.substates ?sub])
      no_chain (linkto [?sub.active nil])))
  ;
# Chain when the "activate_defaults" is set to TRUE
  (and no_chain (?active.type = AND)
    (?active.activate_defaults = true))
  { }
# Reset value and recurse. Set appropriate links
  (and (?active.activate_defaults = false)
    (?sub.activate_defaults = true)
    (link [?active active ?sub]));

```

Figure 7: `activate_default_2` helper rule

### 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

```

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 serially – this implies that it is impossible to generate two events simultaneously; hence events such as “E1 and E2” make no sense. The MSL schema generated by the compiler has a class, `EVENT_LIST`, that contains attributes for each simple event; for the statechart of Figure 1 example, the `EVENT_LIST` class is shown in Figure 3. 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,

```

generate_TIME_FOR_REVIEW[?e:EVENT_LIST]:
{
  (and no_chain (?e.TIME_FOR_REVIEW = false)
              (?e.TIME_FOR_REVIEW = true));
}
-----
generate_TIME_FOR_REVIEW(event)
  Forward: rule_7(event)
  Forward: clear_TIME_FOR_REVIEW(event)
  Forward: reset_states(event)

```

Figure 8: Rule chain for transition from IDLE to REVIEW\_DESIGN

but the compiler does not currently support this. The objectbase has one object `event` (not shown in Figure 2) 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 8. Once the appropriate event has been generated (by the `generate_TIME_FOR_REVIEW` rule), 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 that reset the state of the objectbase for the next event are fired: `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 statechart. Sometimes a condition is described as a compound condition; for example, `APPROVED` is defined in the statechart from Figure 1 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 transition from `DEV_CODE_MODS.IDLE` to `MODIFY_CODE`. If the condition `[CLEAN and APPROVED and CODE_REFLECTS_DES]` were `true`, the system would make an additional transition from `MODIFY_CODE` to `CODE_MODIFIED`. As a second example, the default transition of `DEV_TESTS` activates `MODIFY_UNIT_TEST_PKG` when `[REWORK_TESTS]` is `true`, and `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 (not shown in Figure 2). There are no helper rules for conditions since a condition remains `true` (or `false`) until some other rule changes it.

## 4.5 Actions

Actions are performed once a transition occurs. The following actions are compiled into MARVEL rules:

```

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

```

Actions do not require separate entities in the MARVEL objectbase. `Statemate` allows actions to generate events, but for simplicity, the compiler does not currently support this feature. Actions can cause events and make conditions `true` or `false`, thus enabling transitions in a statechart. The transition in Figure 1, for example, occurs if the system is in the `MODIFY_DESIGN` state and the `wr!(MOD_DESIGN)` action occurs.

## 4.6 History connectors

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

- A state, when exited and re-entered through a history connector, will return to the substate on the same level as the history connector that last had control.

```

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

```

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)

```

```

CONNECTOR
|   initialized : boolean = false;
|   states      : set_of link STATE;
|   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

- A state, when exited and re-entered through a deep history connector, will return to all the states at all levels that 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.

In order to simplify the process of compiling transitions into rules, helper rules are provided that maintain history information. Each history connector is instantiated in the object-base 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`, that determines the states it is maintaining. The `HISTORY_CONNECTOR` and `DEEP_HISTORY_CONNECTOR` are subclasses of `CONNECTOR` (as shown in Figure 10). The *initialized* attribute is `true` when the history (or deep history) connector has been initialized.

Figure 11: Example use of restoring state information from history

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. `rule_3` is the generated rule corresponding to the transition from `MONITOR_PROGRESS` to `HOLD`; it occurs within a forward chain when the `RECOMMEND_CANCEL` event occurs. This forward chain continues to `set_history1` 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 (corresponding to the transition from `HOLD` to `<H*, H*>`) 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, transitions are made to each possible target of the connector (potentially a recursive process). Con-

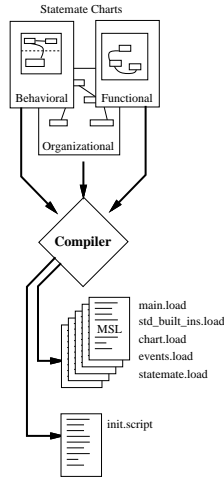


Figure 12: Compiler Architecture

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 [not SUCCESS and not REWORK_CODE]
MODIFY_CODE  [not SUCCESS and REWORK_CODE]
c_0 (Termination) [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 5 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 Compiler

The compiler is written in C and contains 9000 lines of code (20% of which parses the Statechart charts) plus a data structures package of 5000 lines. As shown in Figure 12, the compiler

takes three input files, the State, Module, and Activity charts for a particular process, and produces a set of MSL files and an initialization script, `init.script`. The sample statechart from Figure 1, for example, compiles into 72 MSL rules (of which 24 are standard helper rules) totaling about 1600 lines of MSL. The MARVEL loader then parses the MSL input to create a MARVEL environment. At this point, the data schema is fixed, but the objectbase is empty. The `init.script` is then processed which populates the objectbase with the necessary objects and sets the appropriate links. The final command of `init.script` prepares the environment for use.

## 5 Future work

The compiler we have just described takes three Statechart descriptions – Behavioral, Functional, and Organizational – and produces a MARVEL environment that 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:

- *Incorporate Static Reactions* – In Kellner’s work [8], 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 allow rule chains *between* users (i.e., delegation of tasks). We made some preliminary investigations on the possibility of using Process Weaver [3] that suggest that it could be used to perform the necessary delegation.
- *Incorporate Tools into the Process* – 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 that could be utilized in this fashion.

## Acknowledgements

We would like to thank Marc Kellner and David Raffo for providing the 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.

## About the author

**George Heineman** is a PhD candidate in the computer science department at Columbia University. His research interests include cooperative transactions, software technology, and the intersection of process centered environments with database technology. He received his BA degree in computer science from Dartmouth College, and his MS degree from Columbia University. He is a member of IEEE and ACM. He can be reached at the Department of Computer Science, 450 CS Building, Columbia University, New York, NY 10027, email: heineman@cs.columbia.edu.

## References

- [1] Israel Z. Ben-Shaul, Gail E. Kaiser, and George T. Heineman. An architecture for multi-user software development environments. *Computing Systems, The Journal of the USENIX Association*, 6(2):65–103, Spring 1993.
- [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] Christer Fernström. PROCESS WEAVER: Adding process support to UNIX. In *2nd International Conference on the Software Process: Continuous Software Process Improvement*, pages 12–26, Berlin, Germany, February 1993. IEEE Computer Society Press.
- [4] 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.
- [5] 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.
- [6] Marc I. Kellner. Software process modeling: Value and experience. In *SEI Technical Review*, Software Engineering Institute, pages 23–54. Carnegie Mellon University, 1989.
- [7] Marc I. Kellner. Multiple - paradigm approaches for software process modeling. In *7th International Software process Workshop: Communication and Coordination in the Software Process*, Yountville, CA, October 1991. IEEE Computer Society.
- [8] Marc I. Kellner. Software process modeling support for management planning and control. In Mark Dowson, editor, *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 8–28, Redondo Beach CA, October 1991. IEEE Computer Society Press.
- [9] David M. Raffo. Evaluating the impact of process improvements quantitatively using process modeling. In Ann Gawman, W. Morven Gentleman, Evelyn Kidd, Per-Åke Larson, and Jacob Slonim, editors, *1993 CASCON Conference*, pages 290–313, Toronto, Ontario, Canada, October 1993. IBM Canada Ltd. Laboratory and National Research Council Canada.