Compiling Modechart Specifications*

Carlos Puchol, Aloysius K. Mok and Douglas A. Stuart
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188

Abstract

The Modechart specification language is a formalism for the specification of real-time systems. A toolset for specification, analysis and simulation for Modechart specifications exists for supporting the design and construction of real-time systems [CHLR93]. This paper introduces a new tool in the the toolset: a compiler for a class of Modechart specifications, namely, that of deterministic system specifications, extended by a subclass of the non-deterministic system specifications. The object code that the compiler generates is in ESTEREL*, a member of the synchronous family of programming languages for real-time systems. We discuss a broad approach to the implementation of timing specifications, providing a range of implementation options, from the basic time step unrolling of states in* ESTEREL*, to the use of system timers. The compiler presented herein allows the specifier to obtain a correct implementation of a modechart program, including timing constraints.*

1 Introduction

While defining Statecharts [Har87], Harel already pointed out the similarities between Statecharts and ESTEREL [BG92], such as the synchrony hypothesis, broadcast of signals and instantaneous propagation of events. While the principle of programming in ESTEREL is to control the execution and termination of statements using preemption constructs, the principle of specifying Statecharts is to control the set of active states by entering and terminating them using transitions. Harel pointed out as well one crucial strength of ESTEREL —that compiler techniques are available to generate very efficient implementations of the state machines described by ESTEREL code.

The Modechart specification language [JM94] is a language designed for the specification of real-time systems. It is part of the SARTOR project [Mok85], an effort towards providing an integrated environment for the design, analysis and implementation of real-time systems. Modechart borrows from Statecharts the use of hierarchical graphical formalisms to extend conventional state-transition diagrams. However, the semantics of Modechart is defined axiomatically in terms of RTL (Real Time Logic [JM86]), which is a logic especially amenable to reasoning about the absolute timing of events. A set of tools [CHLR93] has been developed for the specification and analysis of real-time systems within the SARTOR environment, to which we now add implementation capabilities.

In this paper we describe a compiler for the class of deterministic Modechart programs and a subset of the class of non-deterministic Modechart programs. The output of this compiler for a given input modechart is a piece of ESTEREL code, the *object code*, which implements the original specification. The object code in turn is compiled into a finite state machine by the ESTEREL compiler [AGE89]. The translation thus relies on an ESTEREL compiler to produce a state machine in a conventional language such as C or Ada and on a compiler for the latter languages to produce an executable state machine.

The compiler presented herein takes advantage of the compiler developed for ESTEREL, which compiles away communication and concurrency by composing the component state machines; the resulting uniprocessor code is thus compact and efficient. It also takes advantage of the flexibility of combining Modechart code with ESTEREL code, which facilitates the seamless integration of the high-level facilities of Modechart with the excellent low-level facilities of ESTEREL. This provides support for a class of non-deterministic specifications which are necessary in any practical application. Furthermore, this compiler allows the specifier of Modechart specifications to obtain a correct implementation of the specification without manual translation or assistance.

The paper is organized as follows. Section 3 provides an overview of the compiler and the issues involved. The details about the compiler are presented in Section 4. Section 5 shows an example of compilation of a (sub)specification from the popular railroad crossing exam-

*This research is supported by a grant from the Office of Naval Research under grant number N00014-94-1-0582, and also by a grant from the Texas Advanced Research Program. The first author is partially supported by a Fulbright fellowship from Spain's Ministry of Science.

ple. Section 6 carries out a formal proof that the compiler output actually implements the semantics of the language. Section 7 addresses the issues related to the implementation of Modechart timing constraints using ESTEREL and system resources. Section 8 concludes the paper with a discussion of the work presented and directions for future work.

2 Overview of Modechart

Modechart specifications consist of modes, transitions and external events. Modes can be thought of as hierarchical partitions of the state space, and are related by a containment relation, as dictated by how they are syntactically composed (sequentially or in parallel). The basic modes with no children are called *atomic* modes. A *serial* relationship among several modes (children of a serial mode) indicates that the system operates in exactly one of the modes at any time. Several modes can also be in *parallel*, in which case the system operates in all modes simultaneously. A mode is depicted graphically as a box, with all its child modes represented inside it.

Transitions allow the system to switch from one mode to another (possibly in different levels within the hierarchy). Transitions can be triggered by events, which can be external or internal. All the events in the system are instantaneously broadcast and transitions, once triggered, are taken instantaneously. Serial modes may designate one mode as their initial mode, which is instantaneously entered once the serial mode is entered (unless some other child has been entered explicitly with a transition crossing or terminating at it). Transitions between modes in parallel are not allowed. Entry of a parallel mode forces the system to enter all its children. A transition out of one mode requires exit out all modes in parallel with it. A transition is depicted as a directed edge, departing the source mode and ending in the destination mode.

Each transition has a *transition condition* associated with it. There are two types of transition conditions in Modechart, namely *timing conditions* and *triggering conditions*. Timing conditions are those that have the form (lb, ub), where lb is a lower bound and ub is an upper bound. Triggering conditions are of the form $p_1 \wedge p_2 \wedge \ldots \wedge p_n$, where each p_j specifies a condition for taking the transition and can be in one of the following forms: (a) $\{(M_1, M_2, \ldots, M_m)\}$, which requires the system to be in one of the specified modes, (b) $\rightarrow M$ or $M \rightarrow$, which are events corresponding to entering or leaving mode M, (c) $M_1 \rightarrow M_2$, which specifies a mode transition event from mode M_1 to mode M_2, or (d) e or $\overline{e}$, which specifies that the external event e be present or absent respectively. A transition can be taken at time t only if the condition is satisfied at time t.

3 Overview of the Compiler

Modechart and ESTEREL are languages for reactive systems based on a synchronous execution model. One of the main differences between them is that ESTEREL is not a specification language, while Modechart was designed for that purpose. While Modechart provides facilities for specifying a wider range of behaviors, ESTEREL allows only for deterministic programs. Modechart does not provide mechanisms, such as those that ESTEREL provides, for accessing lower level resources such as typed variables and function calls. We argue that ESTEREL is an excellent language for programming reactive systems at small and modest scales, while Modechart is more suitable for medium and large scale systems (in part due to its graphical nature).

Another difference between the two languages is the concept of output of a reaction and the events that shape it. While in ESTEREL an output event is composed of the set of signals emitted during the reaction, in Modechart the output of the system is considered to be the mode transitions of the system from the "state" before the current reaction to the "state" after the reaction has taken place. The state of the system is the set of active modes, active transitions and events in the reaction.

In ESTEREL the events upon which the system evolves are the presence or absence of external and internal signals in the current reaction. In Modechart the events upon which the system evolves are the presence or absence of external signals as well as the current state and its evolution within the current reaction[1].

These differences will become more apparent once we start describing the compilation process, since they shape the structure of the compiler and are crucial in showing the correctness of the process. The Appendix contains a brief description of the ESTEREL programming language.

A crucial factor in the development of the compiler is choosing the right representation for the modes and the transition expressions of Modechart in the ESTEREL object code. Clearly, a mode is a more abstract object than a single signal, since expressions involving modes can refer to the entry of a mode, the exit of a mode or a mode being active. Thus our choice for the representation of modes consists of associating three signals to each mode, denoting respectively the system entering the mode, the system being in the mode and the system exiting the mode. The set of output signals for a given modechart is then the union of all the signals associated with each of the modes. Transition

[1] Note that while ESTEREL allows for control being propagated on the basis of the *absence* of (possibly internal) signals, Modechart does not originally allow so. This simplifies the semantics (albeit slightly limiting the expressibility) of Modechart specifications. We take advantage of this ESTEREL feature by extending the transitions expressions, allowing negated external events (i.e. $\overline{e}$ in case (d) above).

conditions become (in ESTEREL) boolean compositions of signals.

We now briefly describe the syntactic structure of the object code and outline its execution dynamics. Section 4 more formally defines the compiler. For the purpose of this presentation, we omit the details of parsing and type-checking as well as the generation of signal declarations for the object code.

Syntactic Structure Overview. The global syntactic structure of the object code is dictated by the modes in the input specification. Corresponding to the hierarchically organized modes in a specification, the output code is a series-parallel composition of the object code generated for each one of the modes in the specification. The object code corresponding to each transition is placed within the code for the mode of *origin* of the transition (naturally, it can only become active when the code for the mode is active).

The overall syntactic structure of the object code is the following, for an input modechart with modes $m_1, \ldots, m_n$:

$$\text{emit ENTER\_}m_1 \parallel \mathcal{GM}(m_1) \parallel \ldots \parallel \mathcal{GM}(m_n)$$

The function $\mathcal{GM}(m_i)$ represents the portion of the object code corresponding to mode m_i, *as well as* the transitions originating in it. Hence the object code for the whole input modechart is obtained by parallel composition of the object code for all modes in it. The structure of $\mathcal{GM}(m_i)$ is such that it starts executing *in the same reaction* that a special signal for that mode is present ($\text{ENTER\_}m_i$). Assuming m_1 is the root mode of the input specification, a statement of the form $\text{emit ENTER\_}m_1$ is composed in parallel with the rest of the code to trigger the execution of m_1 in the very first reaction of the execution of the system.

Execution Dynamics Overview. The structure of $\mathcal{GM}(m_i)$ is such that *in the same reaction* that m_i is entered, the code for it triggers the entry of all of its children (if the mode is parallel) or at most one of its children (if it is a serial more). Eventually, within the same reaction, all the proper modes are entered. This captures the mode entry semantics of Modechart [JM94, PSM95].

During the first reaction, the root mode, m_1, is entered (i.e. its corresponding code $\mathcal{GM}(m_1)$ is executed in the first time instant), by virtue of the signal $\text{ENTER\_}m_1$ being present in the first reaction only. Still in the first time instant, all the modes that should be entered (according to the semantics) are in fact entered by propagation of the entry signals in one reaction, as explained above, and inside all those modes, within a parallel construct, the code for all the transitions initiating in them is active as well.

As time instants elapse and input events occur, *one or more* of the transitions will eventually be triggered, either by some external event or by the passage of time (more than one transition can be triggered in one time instant, as long as their pairwise least upper bounds, with respect to the containment relation, are parallel modes). We emphasize that this restriction, which essentially enforces determinism, is crucial for assuring the correct execution of deterministic Modechart specifications. This restriction in fact partly determines the class of Modechart specifications supported by the compiler.

The code for each active transition then forces termination of the modes that it (explicitly or implicitly) exits and forces the entry of other modes, the landing modes of the transition [JM94]. This is in essence the execution dynamics of the output code when transitions are taken in one time instant. The process is repeated, with the new set of modes active after each transition is taken.

4 Structure of the Code Generation

A requirement set forth by the compiler is that the input specification be a self-contained hierarchy of modes (and their transitions). All references to external modes (in transition expressions) become references to external signals. The environment of the state machine (through the actual execution driver provided by the user) needs to map these references into the signal domain of the state machine.

The presentation of the actual code generation part of the compiler is done below in a purely functional fashion, that is, the output of the compiler is a string, but it is obtained by (repeatedly) substituting the function calls within the string itself (indicated by an *emphasized* font), until a complete string is obtained. In the definitions below, in the interest of space, we simplify the notation as much as possible and we omit the definitions of some of the functions used; most are related to syntactical declarations for ESTEREL and simple operations on the hierarchy of the modechart, such as *Declarations(), RootMode()* or *ChildrenOf()*.

4.1 Structure of the Modes

For each mode in the specification (and the transitions emanating from it), a self-contained piece of ESTEREL code is generated as its implementation. The code for all the modes is linked by using the ESTEREL synchronous parallel composition operator. Thus for a modechart m, the compiler object code generated, $\mathcal{ME}(m)$, is defined as follows:

$$\begin{aligned}
\mathcal{ME}(m) = \text{module } m\text{:}& \\
&\quad Declarations(m) \\
&\quad \text{signal } \; InternalSignals(m) \text{ in} \\
&\qquad \text{emit ENTER_}RootMode(m) \\
&\qquad GenerateMode(\; RootMode(m)) \\
&\quad \text{end signal} \\
&\text{end module}
\end{aligned}$$

GenerateMode(m)=

```
1 ||
2  loop
3      await immediate ENTER_m;
4      trap QUIT_MODE_m
5              CallAction(m)
6          ||
7              EnterChildren(m)
8          ||
9            [
10             do
11               sustain IN_m
12             watching immediate TERMINATE_m;
13             exit QUIT_MODE_m
14           ]
15         ||
16             TransitionConditions(m)
17     handle QUIT_MODE_m do
18         emit EXIT_m;
19         TerminateChildren(m)
20     end trap
21  end
22  Foreach m_i = ChildrenOf(m) do
23      GenerateMode(m_i)
```

Figure 1: Definition of the function *GenerateMode*.

This embodies the necessary syntactic wrapping to form a valid ESTEREL program. The function *GenerateMode(m)* (in Figure 1) is the principal function that produces the object code for each of the modes (and transitions) enclosed in (and including) m. It can be regarded as the compilation of one mode and all its children. For the sake of simplicity, we define it as a (tail) recursive function (lines 22–23 of Figure 1)[2]. We shall use the concept of "a mode being active" in terms of Modechart or in terms of its ESTEREL object code interchangeably, when not ambiguous.

The three ESTEREL output signals associated with a mode m in the definition of *GenerateMode(m)*, namely $\mathrm{ENTER\_}m$, $\mathrm{IN\_}m$ and $\mathrm{EXIT\_}m$, reflect the "state of activation" of mode m and are emitted at all time instants in which m is entered, m is active or m is exited, respectively. Figure 2 depicts the relation between the "state" of a mode and the presence of its corresponding signals. These signals are output signals by default. The signal $\mathrm{ENTER\_}m$ is present in the time instants that the (code for the) mode m is entered. The signal $\mathrm{EXIT\_}m$ is present in the time instants that m is exited. The signal $\mathrm{IN\_}m$ is emitted at all time instants after (and including) when $\mathrm{ENTER\_}m$ is present up to, but not including, the time instant that

[2] Defining it in this way also helps in realizing the compositional nature of the compiler presented. Note however, that since the order of the generation of the modes is not important, the implementation of *GenerateMode* actually performs the code generation without recursion, i.e., iteratively over the modes.

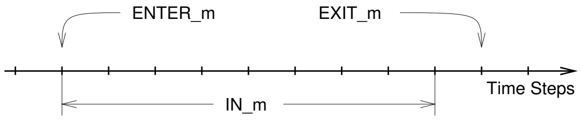

Figure 2: Signals representing the state of activation of a mode.

$\mathrm{EXIT\_}m$ is emitted. If the mode is entered and exited in the same time instant, then $\mathrm{ENTER\_}m$ and $\mathrm{EXIT\_}m$ will be emitted in that reaction, but $\mathrm{IN\_}m$ will not.

The signal $\mathrm{IN\_}m$ is not emitted at the time instant M is exited for practical reasons: it is convenient that no overlapping exists between all $\mathrm{IN\_}m_i$ signals of the children m_i of a mode so that they may have exclusive functionality. The transition expressions corresponding to a mode being active thus check for the presence of $\mathrm{IN\_}m$ or $\mathrm{EXIT\_}m$, which is semantically equivalent to the standard case.

We assume the value returned by *CallActions* is nothing at this point, but in practice it can be some (conceptually instantaneous) procedure call interfacing the environment, or some instantaneous piece of ESTEREL code. Once the mode is entered, zero or more children of the mode are entered (line 7). Three cases arise, depending on the type of the mode. If the mode is atomic, no children are entered. If the mode is serial, exactly one mode must be (explicitly) entered. If no other child mode is active (one of them could have been entered by a transition explicitly terminating at it), the initial child of the serial mode is entered. Finally, if the mode is parallel, all the children are entered simultaneously as the effect of emitting their respective entry signals. Let the function *EnterChildren(m)* be defined as follows, for a given mode m:

EnterChildren(m)=

```
if IsSerial(m) then
    present [ NO_DEF_INITIAL_m ] else
        emit ENTER_ InitialOf(m)
    end present
elseif IsParallel(m) then
    EnterAllChildrenOf(m)
otherwise        // m has to be Atomic
    nothing
```

There is another (internal) signal associated with a mode m, namely $\mathrm{TERMINATE\_}m$. Its purpose is to terminate the execution of the code corresponding to m and (recursively) all its descendants that may be active when it is emitted. This signal is originally emitted by the code corresponding to a transition, as we shall show below, when a transition is taken to transfer control from one mode to another.

When a mode is active, the do-watching construct in lines 10–12 is waiting for the presence of $\mathrm{TERMINATE\_}m$. Once this signal has been received, $\mathrm{IN\_}m$ is not emitted and the $\mathrm{QUIT\_MODE\_}m$ trap exits to the handle construct in line 17. At this point, *within the same reaction*, $\mathrm{EXIT\_}m$ is emitted and a $\mathrm{TERMINATE\_}m_i$ signal is emitted for each child m_i of m. This same process is followed *still within the same reaction*, for all active modes nested in m.

In short, the transformation provided by *GenerateMode* consists essentially of a block of code for each mode. The code for a mode is entered every time instant that the signal $\mathrm{ENTER\_}m$ is emitted. At the same time instant that the mode is entered, the $\mathrm{IN\_}m$ signal is emitted and it is sustained as long as the signal $\mathrm{TERMINATE\_}m$ is not present. When the mode is exited the corresponding children, if active, are terminated as well, as explained above.

4.2 Structure of the Transitions

In Modechart, each active transition can prompt the termination of several modes and the activation of several others by being taken. There are two issues to deal with regarding code generation for transitions. First, the code for each active transition must keep an active statement waiting for the condition of the transition to become true. The condition for the transition is called a transition expression. Second, once the condition for the transition is true, the transition code must determine which modes are exited and which modes are entered by taking the transition.

Waiting for triggering events. The function that generates code to monitor the transition conditions to be awaited is *TransitionConditions*[3]:

TransitionConditions(m)=
    Map TimingTrans TimingTransitionsOf(m)
    Map TriggerTrans TriggerTransitionsOf(m)

The timing expressions the compiler supports are those where the two bounds are equal. The transition must be taken if the bounded amount of time instants has elapsed (unless some other transition is taken first). The evaluation of timing transitions is defined by the following function for a transition t with bound $t.b$:

```
TimingTrans(t)=  || [
                      await tick;   % 1
                      await tick;   % 2
                      ...
                      await tick;   % t.b
                      ResolveTransition(t)
                   ]
```

The second type of transition expressions is triggering expressions: a transition is taken when an event such as entering a mode, or exiting a mode or being in a set of modes occurs. These expressions can be handled by defining the following function (for a transition t, with transition expression *t.mte*):

```
TriggerTrans(t)=
    || [
        if (t.mte = → m_e) then
           await immediate ENTER_m_e;
        elseif (t.mte = m_x →) then
           await immediate EXIT_m_x;
        otherwise
           await immediate [ InModeSet(t.mte) ];
        ResolveTransition(t)
    ]
```

We present a simplified version of the "Otherwise" case to illustrate the function *TriggerTrans*. It is straightforward to see how more complex constructions can support the full set of mode transition expressions defined in Modechart.

Taking a transition. The last remaining task in compiling transitions is determining which modes are to be entered and which are to be exited. The compiler only has to generate code for the emission of the $\mathrm{TERMINATE\_}m$ signal for the outermost mode that the transition exits (we call this mode $LUB-1$) as well as all the entry signals for all the modes being entered by the transition. There is no need for the transition to generate all the termination signals for exiting nested modes because the outermost mode exited will itself generate the $\mathrm{TERMINATE\_}m_i$ signals for all modes contained in m.

Each transition must generate all the entry signals down to the destination mode of the transition because the transition determines the "hierarchy" of the modes that need to be entered. Inside the destination mode, the default entry procedure is followed. The reader should also note that parallel modes can be nested within this path and they will correctly be activated according to the Modechart semantics by the code of the transition. We call these modes $LUB+1, \ldots, LUB+k$. Figure 3 depicts a generic scenario for one transition T.

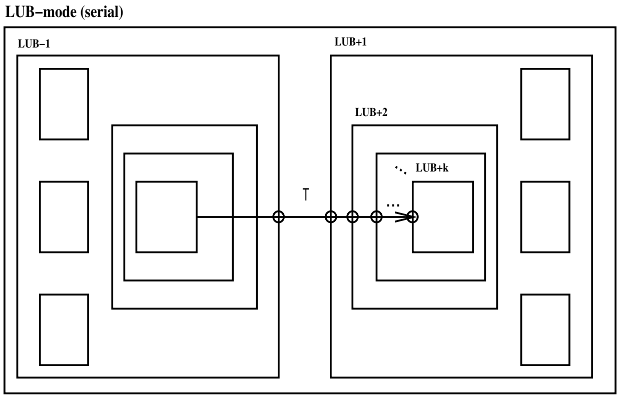

Figure 3: Modes involved in a transition T being taken.

[3] *Map* is a function that, given a function and a list, applies the function to each element of the list and concatenates the results of all calls into a string.

This is done by *ResolveTransition*:

```
ResolveTransition(t)= [
        CalculateLUB(t.from, t.to)
        emit TERMINATE_ t.LUB-1;
        emit ENTER_ t.LUB+1;
        ...
        emit ENTER_ t.LUB+k;
        SetNonDefaultInitials(t.from, t.to)
    ]
```

The function *SetNonDefaultInitials* emits a signal $\mathrm{NO\_DEF\_INITIAL\_}m$ in those cases where m is not the default initial mode of the corresponding serial parent (thus preventing the entry of the initial mode).

Non-deterministic transitions. The class of specifications supported by the compiler so far introduced is the class of those specifications exhibiting deterministic behaviors. The only source of run-time non-determinism is the input signals. However, while this class can be useful, it is of limited applicability in practice, since most real applications have their control structure influenced by the presence of data values, such as contents of messages or bounds on iterations, typically modeled in Modechart specifications by non-determinism. In general, such specifications cannot be compiled into ESTEREL.

To support some of these behaviors, this compiler uses the ESTEREL if conditional statement. Conditionals allow boolean expressions involving globally allocated variables to be evaluated at run time during transitions of the state machine. Thus run-time checks on variables are inserted in the computation tree of reactions.

We introduce support for a limited class of non-determinism by allowing atomic modes to have transitions with predicates on variables (e.g. $\mathrm{ALTITUDE} > 5000$) as transition expressions. The restrictions are that modes that have these transitions cannot have any other type of transitions, and that the conditions must be mutually exclusive and exhaustive. The values of these variables can be arbitrarily changed by code associated with modes. In terms of behavior, these transitions are equivalent to non-deterministic transitions with timing expressions of the form $(0, 0)$.

5 An Example

We show now an example of the compiler output for the popular example of the railroad crossing, namely, the implementation of the gate controller. The input modechart presented in Figure 4 is a refinement of the actual specification in [JS88] to make the timing transitions deterministic.

The following is the corresponding object code output. The declarations of the code have been generated by using an interfacing guide from the user. The gate controller itself

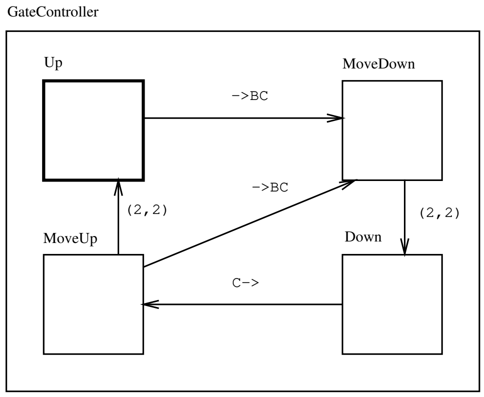

Figure 4: Gate controller of the railroad crossing.

compiles into a 7-state state machine. Due to space limitations, we only show here the code for the root mode and the MoveUp mode (the rest are very similar to the latter). The $\mathrm{ENTER\_BC}$ signal refers to the sensor which detects approaching trains before the crossing. The $\mathrm{EXIT\_C}$ signal refers to a train leaving the crossing. The NI_* signals abbreviate the NO_DEF_INITIAL_* signals defined before.

```
module GateController:
  input EXIT_C, ENTER_BC;
  output ENTER_Up, ENTER_Down,
    ENTER_MoveUp, ENTER_MoveDown;
  relation EXIT_C # ENTER_BC;
  signal IN_GateController, IN_Up, IN_MoveDown, IN_Down,
    IN_MoveUp, ENTER_GateController, EXIT_GateController,
    TERMINATE_GateController, NI_GateController, EXIT_Up,
    TERMINATE_Up, NI_Up, EXIT_MoveDown,
    TERMINATE_MoveDown, NI_MoveDown, EXIT_Down,
    TERMINATE_Down, NI_Down, EXIT_MoveUp,
    TERMINATE_MoveUp, NI_MoveUp in
  emit ENTER_GateController;
||
  loop
    await immediate ENTER_GateController;
    trap QUIT_GateController in
      present NI_GateController else emit ENTER_Up
      end present
    ||
      [ do sustain IN_GateController;
        watching immediate TERMINATE_GateController;
        exit QUIT_GateController
      ]
    handle QUIT_GateController do
      emit EXIT_GateController;
      emit TERMINATE_Up;
      emit TERMINATE_MoveDown;
      emit TERMINATE_Down;
      emit TERMINATE_MoveUp;
    end trap
  end
||
...
||
  loop
    await immediate ENTER_MoveUp;
    trap QUIT_MoveUp in
      [ do sustain IN_MoveUp;
        watching immediate TERMINATE_MoveUp;
        exit QUIT_MoveUp
      ]
    ||
      [ [ await tick;
```

```
        await tick;
        emit TERMINATE_MoveUp;
        emit ENTER_Up;
      ]
    ||
      [ await immediate [(ENTER_BC)];
        emit TERMINATE_MoveUp;
        emit ENTER_MoveDown;
      ]
    ]
  handle QUIT_MoveUp do emit EXIT_MoveUp;
  end trap;
 end
 end signal
end module
```

6 Compiler Correctness

We now concentrate on showing how the object code actually implements the behavior that input specifications describe, according to the semantics of Modechart. Our approach consists of presenting a customized semantics (an interpreter) for the ESTEREL object code.

The conventional behavioral semantics for ESTEREL [BG92] is based on syntactic program transformation. Given a program and an input event sequence, the semantics determines the output event generated by the reaction of the program to the first input event and a new program (to treat the rest of the input). However, using this semantics to show the correspondence between the object code execution and the modechart specification behaviors can be an arduous task that extends beyond the space of this paper, given the intricacy of the object code for even simple specifications.

For this reason, we resort to introducing a customized semantics for the code that the compiler generates, that is, this definition of semantics is only valid for a subset of ESTEREL programs, namely the class of programs that our compiler can generate. This semantics is based on a more traditional concept of state defined by a set of program counters. For simplicity, we concentrate only on pure ESTEREL computations (no conditionals and variables involved).

The ESTEREL interpreter, **EsterelExec** shown below is based on maintaining a set of program counters. Each program counter is associated with a mode m in the specification and can be in one of several positions: m.enter, m.inside and $m.t_1$, ... $m.t_n$, corresponding respectively to the program counter for m being halted in the await statement (line 3 in Figure 1), in the do-watching construct (lines 10–12), or in any of the await statements for transitions t_1 to t_n (expansion of line 16)[4].

[4] For simplicity, we assume that the points for the transitions have information attached regarding the value of the transition expression (for triggering transitions) and time ticks left until the end of the bound has been reached (for timing transitions).

Algorithm: EsterelExec
Input: I (input stream), S (the specification)
Output: O (output stream)
Var: p (program counter set), s (signal set), i (input event), o (output event)

    p := $\{m_1.enter, ...m_n.enter \mid m_i \in S\}$;
    s := {tick, ENTER_m1}; (* assume m1 is the root mode *)
    while true **do**
        (p, o) := **ComputeReaction**(p, s, i);
        Output(o, O);
        PerformStep(p);
        i := read(I);
        s := {tick};
    end while
end algorithm

Function: ComputeReaction
Input: p (program counter set), s (signal set), i (input event)
Output: p (program counter set), o (output event)
Var: pc (program counter)

    o := $\emptyset$;
    while (pc := PickPoint(p, s, i, o)) **do**
        case pc of
        m.inside:
            o := o $\cup$ IN_m;
            if (TERMINATE_m $\in$ s) **then**
                o := (o $\cup$ EXIT_m) - IN_m;
                s := s $\bigcup_{c \in m}$ TERMINATE_c;
                p := p $-$ m.enter;
                p := p $- \bigcup_{tr \in m}$ m.tr;
            end if
        m.enter:
            if (ENTER_m $\in$ o) **then**
                o := o $\cup$**EnterChildren**(m);
                p := p $\cup$ m.inside $\bigcup_{tr \in m}$ m.tr;
            end if
        m.t:
            if ((m.t is timing and **BoundReached**(m.t)) orelse **Triggered**(m.t, o) **then**
                (orig, dest_list) := **LUB**(m.t);
                s := s $\cup$ TERMINATE_orig;
                o := o $\bigcup_{d \in \texttt{dest\_list}}$ ENTER_d;
                o := o $\cup$ **NonDefaults**(m.t);
            end if
    end while
    return(p, o);
end function

The interpreter repeatedly picks an input and computes an output while performing the reaction in **ComputeReaction**. The variable p is the set of program counters or points, s is the set of internal signals, i is the current input and o is the current output. **ComputeReaction** repeatedly picks "active" points (i.e. not halted in an await statement) and performs actions according to the type of point, until no more "active" points are left, i.e., a fixed point has been reached by the program. **ComputeReaction** then terminates, returning the set of points and the computed output.

The function **PickPoint** fails when none of the available points can progress, i.e. when all of them have reached a fixed point, or in other words, when none of the conditional tests is true. **PickPoint** selects all points in priority: first the possibly terminating modes (m.inside), then the mode entries, and finally the transitions. The function **PerformStep** takes all the points that are awaiting on the tick signal and moves them one step further. This may trigger one or more transitions. The rest of auxiliary functions are similar to some of the functions in the translation above.

The next step is to prove that this specialized ESTEREL semantics in fact implements the semantics of Modechart by equating this semantics to the operational semantics of Modechart [PSM95]. The operational semantics defines a two-level Kripke structure on which lower level "steps" in the transition system correspond to instantaneous reactions and the top level steps correspond to individual time instants. In both structures the *state* in an instant of a reaction is modeled as a tuple consisting of the active modes and transitions in the system at that instant, plus the output computed thus far in the reaction. For the equivalence theorem below, we build the state of the operational semantics by selecting from the set of program counters those corresponding to active modes and active transitions, along with the current output, thus establishing the link between them.

Theorem 1 *Let S be a deterministic, zero-cycle free modechart. Consider the compiled (* ESTEREL*) version of S and its interpreter as defined above. Let p be the the set of program counters and o the output set resulting from* **ComputeReaction.** *Let M be the set of modes such that $m \in M$ iff $m.inside \in p$. Let T be the set of transitions such that $t \in T$ iff $m.t \in p$ for some m. Consider the triple (M, T, o) as a configuration of S (according to the operational semantics). Then, given the same input I, each interpreted execution of S and each step in the transition system produce the same output.*

Proof outline: We need to consider the outcome of a reaction to a given input, thus we concentrate on the **ComputeReaction** function. First, the output o of the reaction is empty. The function **PickPoint** picks one point off p every time it is called. It picks non-halted points of the form m.inside and m.enter first, if any, assuring all mode exit and entries as consequence of transitions are in fact carried out before any other transition can be taken. Giving priority to the mode exits ensures all the exits are done first, then when no more exits are needed the mode entries take priority, which in essence performs the equivalent function of the "closure" operator of the operational semantics.

When all "mode points" remain halted, one triggered transition is taken, if any. The results of this transition being taken are a number of modes being exited and entered. This equivales to a single transition in the "reaction" structure of the operational semantics.

Given that S is deterministic, no matter the order in which the transitions are taken, no triggered transition can be prevented from being taken. Furthermore, since S has no zero-cycles, no mode can be entered or exited more than once (hence no transition can be taken more than once either). Because of this and the fact that S is finite, the loop in **ComputeReaction** must terminate (in at most $O(2|\mathcal{M}| + |\mathcal{T}|)$, where $\mathcal{M}$ is the set of modes and $\mathcal{T}$ is the set of transitions in S).

It is straightforward to see by comparison that the signals added to the output o and the points added to p in each step are the corresponding ones as the ones added to the state in the operational semantics, therefore, each iteration involving a transition (until another is taken or the loop terminates) is consistent to one step in the reaction relation of the operational semantics, thus the outputs are equivalent. ■

Theorem 1 stipulates that externally, the output sequence of the interpreted execution of S and the output sequence computed by the operational semantics are equivalent, for a given input sequence I, i.e. $O = S(I)$, where $S(I)$ is the corresponding output sequence for I in the execution of S computed according to the operational semantics. The proof of Theorem 1 is based on the interpreted semantics for a translation schema from Modechart to ESTEREL that is slightly more complex than the one shown in this paper; we omit said translation schema due to space limitations.

The additional issues addressed by the extended translation are related to the details in supporting special cases. One such special case arises when a mode can be *entered and exited* in the same time instant (e.g. a mode with an outgoing $(0, 0)$ timing transition). In this case, replacing the loop-and-await construct in lines 2–3 of Figure 1 with a statement of the form $\mathrm{every\ immediate\ ENTER\_}m\ \mathrm{do}$ would solve the problem. However, this solution would not work with modes which can *exit and enter* in the same time instant (e.g. modes with self loops). A general (but less efficient) solution for all cases is to duplicate the code inside the loop construct. The problem of *exactly* detecting which implementation is best for a mode is in general difficult (NP-hard).

7 Implementing Timing Specifications

The notion of physical time ESTEREL is replaced by a notion of ordering among events, with no absolute relationship among them, where time can be handled as an external signal just like any other signal. This is called the multiform notion of time in the ESTEREL context. The problem of accurately relating events to physical time is avoided altogether and pushed to the implementation. In order to achieve a correct behavior with respect to time, the system must relate its events to physical time in some way.

Since Modechart specifications can specify relative and absolute relationships between passage of time and changes in control within the modes of the specification, we have explored different ways of establishing this relationship and how to achieve correct behavior for specifications with an ESTEREL implementation. We present three approaches supported by the compiler presented here. The first one, *dense timing,* is an inefficient approach based on "unrolling" the timing specifications by creating one state per time unit to be waited. This can be used in limited circumstances. The natural extension to this it to use the ESTEREL *counting occurrence,* which is more efficient, but still not as efficient and precise as the third approach, using *operating system timers* (specially for large time values). Using timers is not an option for smaller time values, however, owing to the overhead of using them.

Our compiler supports these approaches by establishing two (ordered) thresholds, which can be changed by the user: if the time specification is at most the small threshold, the first approach is used. Else, if the value is at most the second threshold, the counting occurrences are used. Otherwise, using operating system timers is the choice. Additionally, the user can select the counter or timer implementations for individual transitions by specifying (counter b) or (timer b) as the transition expression.

Dense Timing. In ESTEREL, there is a special built-in signal called tick, which is present at all time instants. The statement await tick waits one time instant. If the state machine generated by the compiler is invoked periodically (and the period is fixed and known), then awaiting for the tick signal can be used to time events.

Thus the first approach is to use the same "time base" in Modechart as in ESTEREL, and translate references to units of time in Modechart into finite sequences of wait statements over the signal tick. Obviously, this approach does not provide an efficient implementation, for two reasons. One is that it's an "active wait," that is, the state machine needs to be executed when any timing is in effect, the other is that the await statements can produce a potential blowup in the number of states of the final state machine (due to parallel composition of modes).

However, the introduction of minimal time delays to prevent specifications from exhibiting "zero loops" is fairly common. In these cases, translating an (alarm 1) condition into an ESTEREL await tick is the right translation. One advantage of using this approach is that it does not prevent formal reasoning to be performed over the final automaton, including timing properties.

Timing with Counters. This approach consists of replacing a sequence of await statements of length b in by a statement of the form await *b* tick;. This is effectively translated by the ESTEREL compiler itself into *one state* and *one counter,* rather than *b* states. This approach solves the potential state blowup problem from above, but not the active wait problem. This approach can be useful when the delay is not large enough to justify using the (potentially) more expensive approach presented next, using system timers. It also requires that the period at which the state machine is called be fixed and known to achieve the behavior specified.

Efficient Timing. As the amounts of time to elapse grows larger, it becomes apparent that it is much more efficient and precise to use the underlaying facilities of the system (both operating system and hardware), such as timers and alarms to take care of timing events instead of counting time steps. Obviously, this approach is particularly useful with larger delays (relative to the execution period of the state machine). This approach is required when the state machine invocations are not guaranteed to be regular in period or no relation between time ticks and physical time can is guaranteed. Our last approach thus is to provide an interface for timers as shown below, for transition t with bound $t.b$.

TimingTrans(t, b) = [
    emit TIMER_SET_t(*b*);
    await TIMER_EXPIRED_t;
    ResolveTransition(t)
]

Every alarm timer, t, allocated in the system is interfaced through the signals TIMER_SET_t(long), TIMER_EXPIRED_t and TIMER_CLEAR_t associated to it. A separate ESTEREL module for timers is generated, where all the timer interfaces run in parallel. The timer module itself runs in parallel with the whole system.

The semantics provided by these signals is as follows: the SET signal is emitted by the *user* of the timer to (re)start the *one-shot* timer t. This is a signal of type long, which indicates the number of time units that the timer is set. The CLEAR signal is also used by the user of the timer and the purpose is to cancel an active alarm. The EXPIRED signal is emitted by the timer module to indicate the expiration of the timer.

8 Discussion and Future Work

We have presented the first compiler for Modechart programs. It supports the class of deterministic Modechart programs and a subset of the class of non-deterministic specifications. The compiler output code is in ESTEREL, a programming language for reactive systems that is based on the synchronous execution model. An ESTEREL compilation step is first required to obtain a state machine in a conventional programming language which is in turn compiled to obtain an executable implementation of the state-machine specification. As long as the ESTEREL and the conventional language compilers are correct (or trusted), a correct (or trusted) implementation of the input specification is accomplished.

The code generation step takes time and space proportional to the size of the input (number of modes plus transitions). The compiler offers the possibility of combining ESTEREL code within Modechart specifications to obtain maximum benefit of the specification capabilities of Modechart and the low level facilities of ESTEREL. It offers support for a choice of implementations for timing specifications, which is generally absent in other compilation of state machines. Our compiler is written in the SML programming language. It can generate a graphical user interface front-end (in tcl/tk) for the state machine so that it can be animated and visualized in action.

Several areas of development are under consideration for the compiler presented here, including development of techniques for detecting non-deterministic specifications and modes with possible instantaneous entry-exit and/or exit-entry behaviors, support for the action model and stronger type checking. In the interest of supporting a larger class of specifications, we plan to support as well priorities among transitions (an extension to the RTL semantics offered in the operational semantics). Many practical non-deterministic specifications become deterministic if priorities are allowed among transitions, therefore significantly enlarging the class of specifications supported by the compiler.

References

[AGE89] AGEL workshop manual version 3.0, 1989. Produced by ILOG, Mountain View, CA, USA.

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[CHLR93] P. C. Clements, C. L. Heitmeyer, B. G. Labaw, and A.T. Rose. “MT:A Toolset for Specifying and Analyzing Real–Time Systems”. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 12–22, December 1993.

[Har87] D. Harel. “Statecharts: A Visual Formalism for Complex Systems”. *Sci. of Computer Programming*, 8:231–274, 1987.

[JM86] F. Jahanian and A.K. Mok. “Safety Analysis of Timing Properties of Real-Time Systems”. *IEEE Transactions on Software Engineering*, 12(9):890–904, September 1986.

[JM94] F. Jahanian and A. Mok. Modechart: a specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, December 1994.

[JS88] F. Jahanian and D. Stuart. A method for verifying properties of modechart specifications. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1988.

[Mok85] A. Mok. SARTOR –a design environment for real-time systems. In *Proceedings 9th IEEE COMPSAC*, 1985.

[PSM95] C. Puchol, D. Stuart, and A.K. Mok. An operational semantics for Modechart specifications. Technical Report UTCS-TR95-37, Department of Computer Sciences, The University of Texas at Austin, September 1995.

Appendix: ESTEREL Overview

In the ESTEREL programming model, a program specifies a set of modules that run (synchronously) in parallel. Modules communicate with one another and the outside world through *signals*, which are broadcast and may carry values of arbitrary types. Consistent with the synchrony hypothesis, the emission and reception of signals is considered to be instantaneous.

ESTEREL programs allow only deterministic behaviors: the inputs to every reaction (and the current values of variables) fully determine the outputs emitted in that reaction as well as the input-output behavior of the rest of the program. Along with the synchrony hypothesis, both communication and pre-emption preserve determinism. Furthermore, all internal communication is compiled away, and a single deterministic finite state machine is generated by the ESTEREL compiler. This state machine is obtained by first translating the ESTEREL programs into an intermediate code that compiles away the control structure while preserving concurrency. Then the code is symbolically executed to extract the automaton. Thus the parallelism in ESTEREL is a structuring tool for programming convenience, and does not incur any run-time overhead — the compiler automatically performs the complex interleaving between parallel modules. Furthermore, since the final implementation is a finite state machine, the time taken by any reaction can be accurately bounded if the execution times of the transitions are known.

Language Constructs. The emit S statement, where S is a signal, indicates that S is present in the current reaction. The sustain S construct is equivalent to emitting S forever. The await S construct blocks until the next reaction in which S is present. The await immediate S triggers if S is present in the current reaction; otherwise, it blocks until the next reaction in which S is present.

The present S then B1 else B2 end construct checks whether the signal S is present in the current reaction; if so, B1 is executed, otherwise B2 is executed. The if P then B1 else B2 end construct evaluates the predicate P *at run time* and if true, B1 is executed, otherwise B2 is executed. The every S do B end construct restarts B in every future reaction in which S is present. The every immediate S do B end construct restarts B in every reaction, including the current one, in which S is present. The do B watching S end construct executes B until the next reaction in which S is present. The do B watching immediate S end construct executes B until the first reaction, including the current one, in which S is present. Finally, the loop B end is an infinite loop, in which B is executed continually.