Design Patterns of Communicating Extended Finite State Machines in SDL*

YoungJoon Byun and Beverly A. Sanders
Computer and Information Science and Engineering
University of Florida
Gainesville, FL 32611
{ybyun, sanders}@cise.ufl.edu

Chang-Sup Keum
Network Technology Lab.
ETRI
161 Kajong-Dong, Yusong-Gu, Taejon 305-600, Korea
cskeum@etri.re.kr

1. Abstract

A design pattern provides a generic solution for recurring problems. Thus a design solution that has worked well in a particular situation can be used again in similar situations. This paper will present several design patterns for communicating extended finite state machines that are useful to model the behavior of telecommunications systems. We will describe each state machine in a state transition diagram and implement the diagram in the Specification and Description Language (SDL), a formal description language for communicating systems.

The patterns presented in this paper are focused on the behavioral aspects of the telecommunications systems. We will begin with a basic pattern of a communicating extended finite state machine and its extension that supports condition and timing constraints in finite state machine. Then we will show the composition of the basic patterns. After that, we will present a pattern that handles repeated events with timing constraints. Finally, we will present a pattern that is used for the creation of dynamic entities in client-server environments.

2. Introduction

When programmers develop software systems, they often find many similar situations that happened in previous developments. A design pattern provides a generic solution for the recurring problems [Gamma+95][Buschmann+96]. This is one of the motivations for using design patterns in software development. In other words, a design solution that has worked well in a particular situation can be used again in similar situations in the future. Design patterns, therefore, can improve software quality and reduce development time through experience, reusability, and modularity of patterns.

In the early development phases of reactive systems such as telecommunications systems, designers model the desired behavior of a system at an abstract level, and then implement the model in an implementation language. Complex behavior can be obtained by combining simpler building blocks: receiving an event from outside, performing a computation in response to the event, and generating output events.

In this paper, we present patterns that we hope will be helpful to a telecommunications systems designer in the early design phases. The patterns describe behavioral patterns both informally and using a formal description technique, communicating extended finite state machines (CEFSMs). CEFSMs consist of a finite set of states and transitions between the states. They communicate with each other and the environment via telecommunications paths through which signals flow. A CEFSM moves from one state into another by an input event while performing actions during the transition. As

---

This work was supported by a grant from ETRI.

described in our first pattern, it may have predicates to select a transition for an event, and timers to handle timing constraints.

The patterns also show the translation of the CEFSMs into SDL [ITU-T96]. SDL is an international standard language to specify telecommunications systems, and is also used in complex, event-driven, real-time, and interactive systems where many concurrent processes communicate each other by using signals. SDL is able to describe the structure, behavior, and data of target system with a mathematical rigor that eliminates ambiguities and guarantees system integrity [Ellsberger+97].

In the following sections, we will present four patterns for telecommunications systems. They are based on the CEFSMs, and each pattern models common behavior of CEFSM in a state transition diagram. In Section 3, we will show a basic CEFSM that describes telecommunications systems formally. We will extend the pattern with predicates in order to control the behavior of a CEFSM. The predicates help to reduce the number of states of CEFSM. We will also present a pattern with timers to handle the timing constraints. In Section 4, we will propose three ways to expand the basic patterns by merging the states of the basic CEFSMs. After that, we will present a pattern that handles repeated events with timing constraints. The pattern is a combination of predicates and timers. The last one will present the dynamic creation of entities in telecommunications systems. We will conclude by representing related work and further research.

3. Communicating Extended Finite State Machine

Context

Many telecommunications systems react to the events coming from the outside environment. The systems can be modeled by distinct states and transitions. When a system receives an event, it moves from its current state to a new state while performing some actions and providing output signals.

Problem

How can we formally model the telecommunications systems?

Forces

Understandability: We need to be able to describe the events, actions, and states in a standard notation to be easy for communication with others. If the pattern is written using a standardized formal description, a tool support such as analysis, simulation, verification, and code-generation might be easy to apply.

State Explosion: In representing the configuration of a telecommunications system, the number of states of the system can increase rapidly such that it is difficult to model the system, which is known as *state explosion*. For example, if a system implements a function of an eight bit counter, it needs at least 256 (=2^8) states to represent each value of the counter. It is necessary to represent the configuration in a contracted form.

Timing Constraints: A reactive system such as a telecommunications system is an event-driven system, which means an event initiates actions for the event. Sometimes the event may occur later than it is expected, or not happen at all because of transmission delay, heavy traffic, wrong specifications, and so on. We must impose timing constraints in modeling to handle the delayed event or the infinite waiting.

Easy Implementation: The system that is modeled by the formalism should represent the system's behavior clearly and precisely. After that, developers implement or translate the model into a specific target language for the execution of the behavior. It is necessary for the formalism to provide a simple path from model to implementation.

Solution

The behavior of a telecommunications system can be described in a communicating extended finite state machine (CEFSM) that is composed of states and transitions among them [Ellsberger+97][Rozenblat01]. For a transition, the system must receive an event from environment, and then it performs corresponding actions for the event. After the actions, the system emits output signals if it has something to notify to the environment. In this paper, we consider deterministic behavior, which means that the system has only one next state for an event. It is unusual to design a system in one CEFSM. A system is generally composed of several entities, and each entity is designed in a CEFSM. In this paper, we use *entry* and *CEFSM* interchangeably.

A CEFSM is 6-tuple: $CEFSM = (S, s_0, E, f, O, V)$, where S is a set of states, s_0 is an initial state, E is a set of events, f is a state transition function, O is a set of output signals, and V is a set of variables. The function f returns a next state, a set of output signals, and action list for each combination of a current state and an input event. For example, $f(S_1, e_1) \rightarrow (S_2, \{o_1, o_2\}, (encoding))$ describes that upon an event e_1 of an entity in a state S_1, it performs an action *encoding* and outputs o_1 and o_2 while moving to the next state S_2. Although a general CEFSM doesn't have a field for the action descriptions, we add the field to briefly describe the main activities of a transition. The actions will be refined in later develop phases. The variables are used to keep the values that are also used in predicates, which will be described later.

A CEFSM can be represented by a state transition diagram, a directed graph whose vertices correspond to states and whose edges correspond to transitions. Figure 1 shows a basic CEFSM and its state transition diagram in which there are two states, S_1 and S_2, and two transitions. The arrow without a source state points to the initial state S_1, and the arrow indicates the initial transition describing the actions and output signals during initialization. Transitions that do not alter the state are indicated by an arc that points to itself. The transition is labeled with an event, action list, and output signals. It is denoted by *event (parameters)/actions/outputs*. The event may have parameters passing information from other entities. Note that all events except the event of initial transition are mandatory while actions and outputs of transitions are option. The '-' symbol in a transition indicates that there is no corresponding value at that field.

Basic_CEFSM = ({S_1, S2}, S_1, {event}, f, {$outputs_1$, $outputs_2$}, {param})
, where *f* is defined as
f(-, -) → (S_1, {$outputs_1$}, ($actions_1$)) /* initial transition */
f(S_1, event) → (S_2, {$outputs_2$}, ($actions_2$))

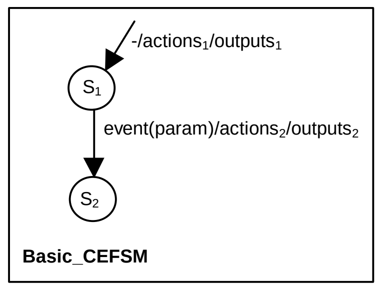


Figure 1. Basic CEFSM and its State Transition Diagram

Predicates

A CEFSM can have predicates to control the behavior of the CEFSM so that some similar states can be grouped to reduce the total number of states [Ellsberger+97]. Upon receiving an event, the machine checks a predicate that is composed of variables, logical operators such as *AND*, *OR*, and comparison operators such as *equal to, less than,* and *greater than*. If the predicate is true, the entity performs actions and produces output signals if it has something to notify to the outside entities. In short, the CEFSM with predicates decides its next behavior based on the predicates.

The CEFSM with predicates is also 6-tuple, $CEFSM = (S, s_0, E, f, O, V)$, as the basic CEFSM except the state transition function. The function *f* returns a next state, a set of output signals, and action list for each combination of a current state, an input signal, and a predicate. The predicate is a pre-condition for the function execution. The action may include an assignment to a variable and a mathematical operation on the variables. For example, $f\,(S_1, e_1, counter = 8) \rightarrow (S_2, \{o_1, o_2\}, (counter{:} = 0; encoding;))$ describes that if an entity in a state S_1 receives an event e_1 and the value of variable *counter* is eight at that time, it will move to the next state S_2 and outputs o_1 and o_2 after setting the *counter* to zero and performing *encoding.*

Figure 2 shows a CEFSM with predicates and its state transition diagram. A transition is labeled with an event, a predicate, action list, and output signals. It is denoted by *event (parameters)[predicate]/actions/outputs*. Note that each predicate that must return a Boolean value *TRUE* or *FALSE* is composed of variables and logical operators. The variables in a predicate are generally either local variables of the entity or parameters of event. The predicate is an optional field. If a predicate field is empty, it is the same as the basic CEFSM. It is very often to have several predicates for an event. For example, if $event_1$ and $event_2$ of Figure2 are the same, we have two predicates, $predicate_1$ and $predicate_2$, for the event. In that case, the control goes to a different transition depending on the result of predicates. To guarantee the deterministic behavior of an entity, the predicates for an event must be mutually disjoint.

CEFSM_Predicates
= ({S_1, S_2, S_3}, S_1, {$event_1$, $event_2$}, f,
{$outputs_1$, $outputs_2$, $outputs_3$}, {$param_1$, $param_2$})
, where *f* is defined as
f(-, -, -) → (S1, {$outputs_1$}, ($actions_1$)) /* initial transition */
f(S_1, $event_1$, $predicate_1$) → (S_2, {$outputs_2$}, ($actions_2$))
f(S_1, $event_2$, $predicate_2$) → (S_3, {$outputs_3$}, ($actions_3$))

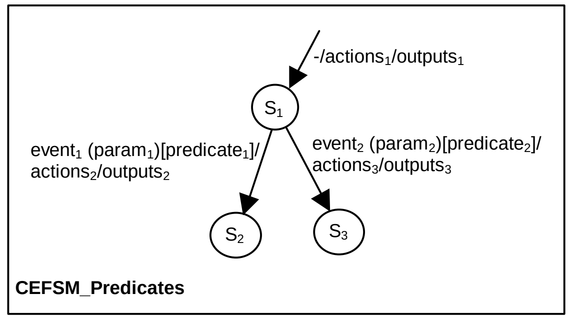


Figure 2. CEFSM with Predicates and its State Transition Diagram

Predicates after Actions

In the CEFSM with predicates, an event is followed by predicates to decide next transition. But in some cases decisions need to be made after performing some actions. In other words, after performing a sequence of actions for an events, an entity decides its next transition depending on the result of the actions. So the entity needs predicates after the actions. Figure 3 shows a state transition diagram for this type of CEFSM in which there are four states, S_1, S_1', S_2, and S_3. Note that the $actions_2$ and $outputs_2$ of the transition from state S_1 to state S_1' is followed by the predicates $predicate_1$ and $predicate_2$. The control goes to a transition depending on the result of predicates. Note also that the transitions from S_1' do not have an event field.

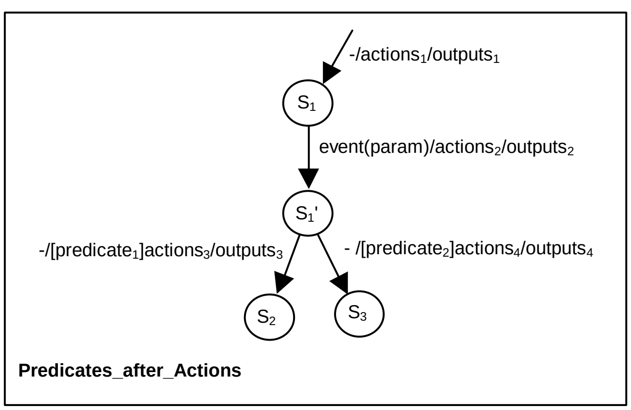


Figure 3. State Transition Diagram of CEFSM with Predicates after Actions

Timers

A CEFSM can be supplemented with timers and timer-related operations. During a transition, the entity sets a timer with a time value. If the timer is not cancelled by the entity, the timer will generate a time expiration signal after the time duration has been exceeded. Generally, the entity handles the time expiration by either sending an error notification or requesting a resubmission of the event. When an event that is wanted by an entity occurs before the time expiration, the entity cancels the timer and proceeds normally. There are two timer-related operations, *set* and *reset*. *Set (v, T_1)* associates a time value *v* with a timer T_1. *Reset (T_1)* cancels the associated timer T_1. We assume that the time unit, for example milliseconds, is not defined in the pattern. It depends on the context of the application. Actually, SDL does not define the time unit [Ellsberger+97].

The CEFSM with timers has the same definition as the basic CEFSM except it includes timers and operations for the timers. Timer is an element of variable set. Figure 4 shows a CEFSM with a timer and its state transition diagram. During the initial transition, a timer T_1 is set with a time value *v*. On timeout for the timer T_1, the entity moves to the state S_2 performing actions *actions*$_2$ and outputting signals *outputs*$_2$. If the entity receives an event *e* that is wanted by the entity, it resets the timer and then performs the remaining transition.

CEFSM_Timers = ({S_1, S_2, S_3}, S_1, {e}, f, {outputs$_1$, outputs$_2$, outputs$_3$},
{param, v, T_1})
, where *f* is defined as
f(-, -) → (S_1, {outputs$_1$}, (set(v, T_1), actions$_1$)) /* initial transition */
f(S_1, T_1) → (S_2, {outputs$_2$}, (actions$_2$))
f(S_1, e(param)) → (S_3, {outputs$_3$}, (reset(T_1), actions$_3$))

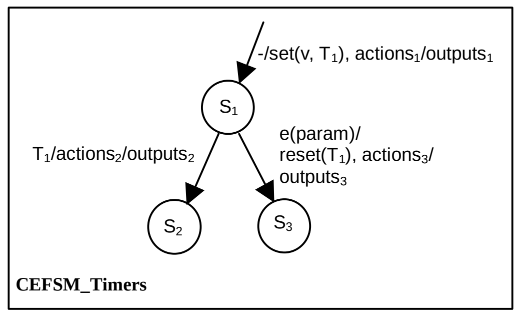


Figure 4. CEFSM with Timers and its State Transition Diagram

Translation into SDL

The translation of a CEFSM into SDL can be done by a mechanical one-to-one mapping. Figure 5 shows an SDL diagram fragment for the basic CEFSM of Figure 1. The initial transition is converted to a start symbol and a state

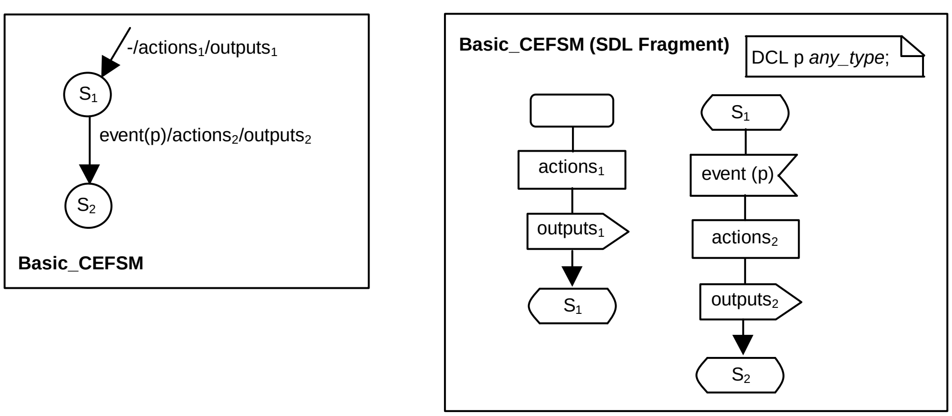


Figure 5. Translation of Basic CEFSM into SDL

symbol S_1. The initialization actions *actions*$_1$ and output signals *outputs*$_1$ are located between them. A transition is represented by a input signal, tasks, and a sequence of output signals of SDL. In the figure, when the state S_1 receives an input *event* with a parameter *p*, it performs a task *actions*$_2$ and outputs *outputs*$_2$.

Translation of Predicates

The translation of predicates needs the *decision* symbols. Figure 6 shows an SDL diagram fragment for a CEFSM with predicates where there are two predicates for one event. Upon receiving an *event*, the entity checks the predicates $predicate_1$ and $predicate_2$ for the event. If the $predicate_1$ is true, it performs $actions_2$ and emits $outputs_2$, while if the $predicate_2$ is true, it executes the action $actions_3$ and generates $outputs_3$.

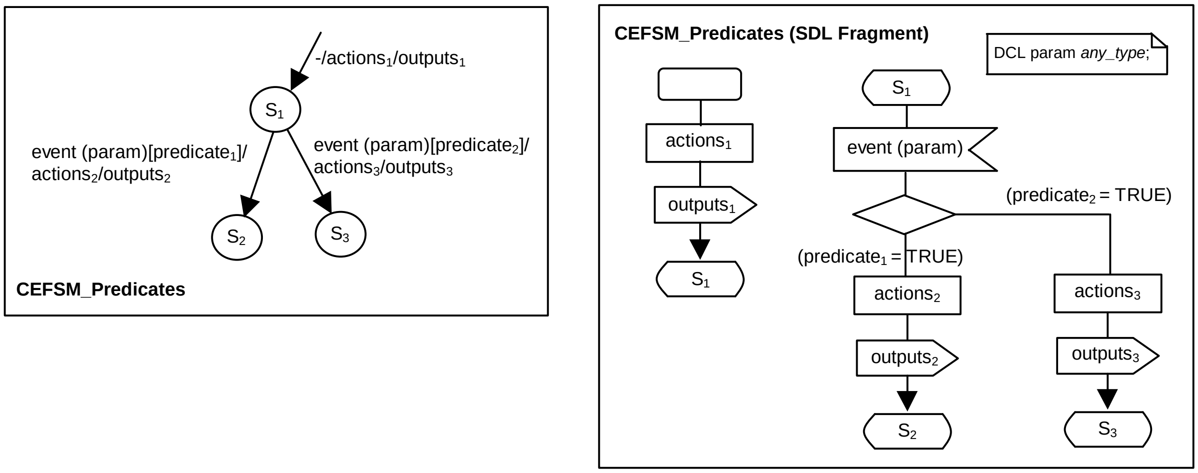


Figure 6. Translation of CEFSM with Predicates into SDL

Translation of Predicates after Actions

The translation needs the *decision* symbols after actions and outputs. Figure 7 shows an SDL diagram fragment for the predicates after actions. Upon receiving an input *event*, it performs $actions_2$ and $outputs_2$, and then the entity checks the predicates where the predicates may be affected by the $actions_2$. Note that the state S_1' is not shown on the SDL diagram.

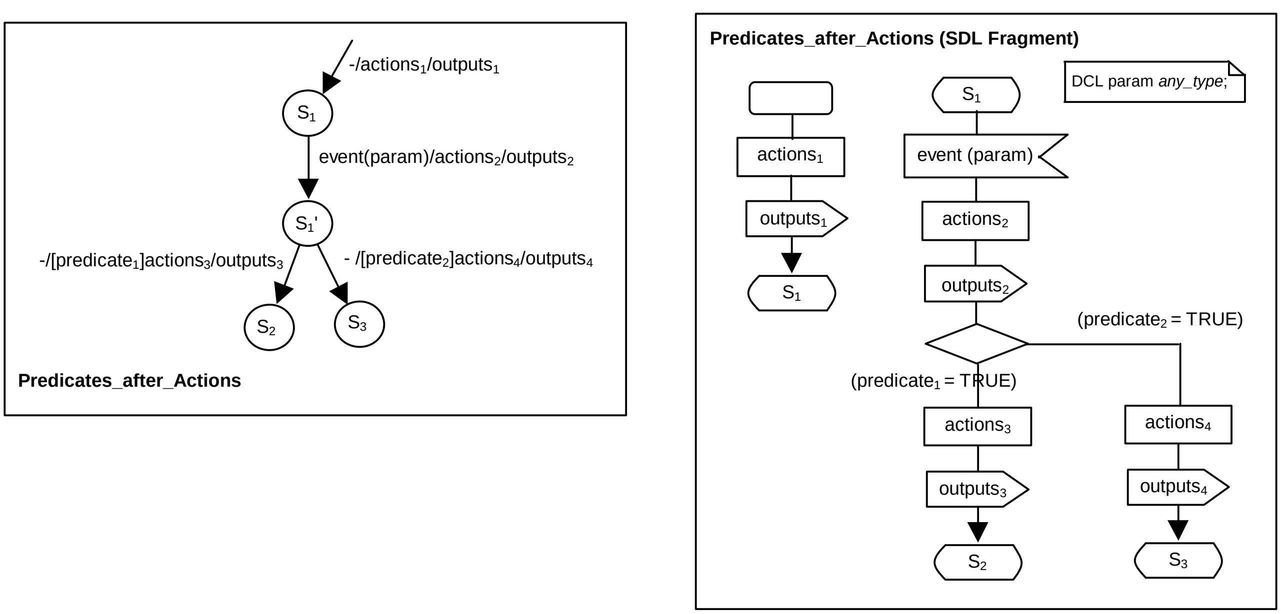
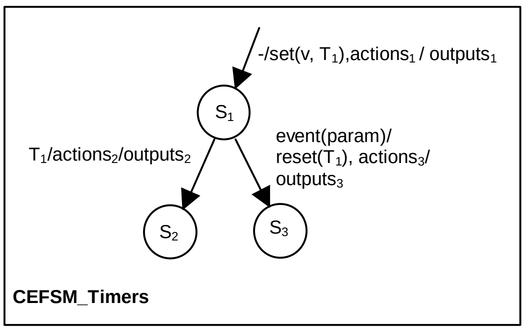


Figure 7. Translation of Predicates after Actions into SDL

Translation of Timers

Timers must be declared before they are used as other variables. Timers are objects to generate timer signals in SDL. Figure 8 shows an SDL diagram fragment for the CEFSM with timers. When a timer T_1 is declared, a timer value *v* is assigned. So we don't need the value in *set* operation. By the set operation, the timer is activated and can generate a timer signal after the time value. In the state S_1, timer signal T_1 is handled as other events. If the event *event* happens in time, the timer T_1 must be reset to deactivate the timer.



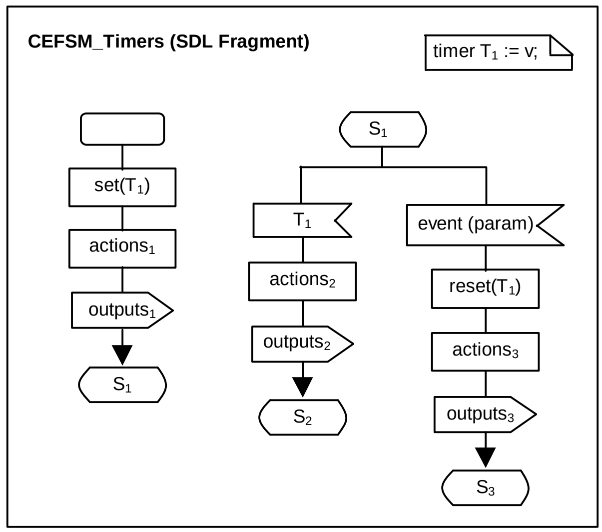


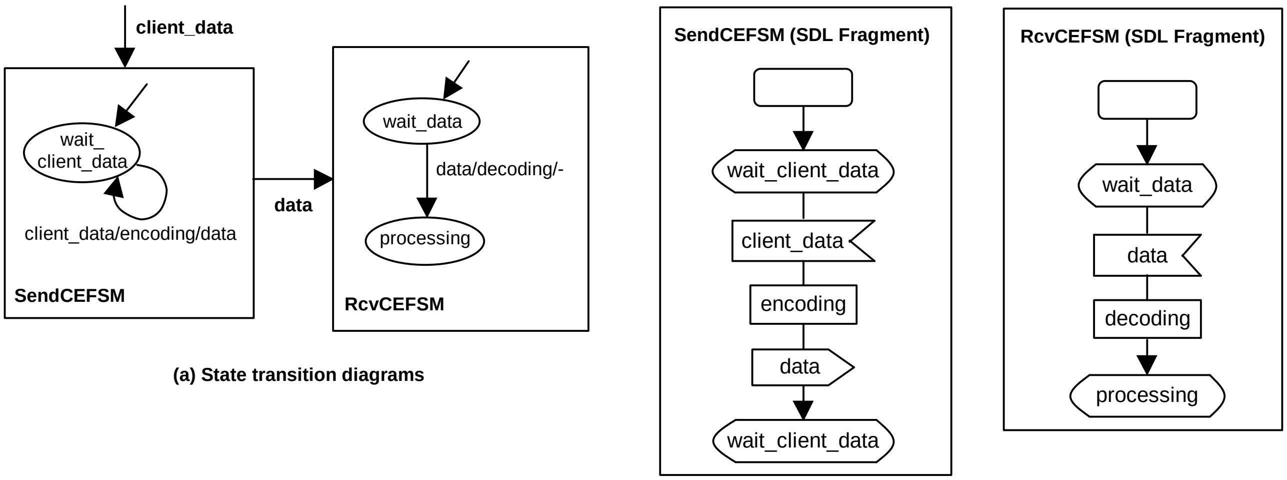
Figure 8. Translation of CEFSM with timers into SDL

Example

The behavior of a telecommunications system is described by a set of processes in SDL. A process corresponding to a CEFSM communicates with other processes by signals. Each process has an implicit input queue where signals are buffered in a FIFO style. The arrival of a signal triggers a transition, and the process can then execute a set of actions such as variables computation, procedures calls and signals outputs. The example illustrated in Figure 9 shows a data transfer between two basic CEFSMs that cooperate to send a data from one CEFSM to another CEFSM.

The sending entity *SendCEFSM* receives an event *client_data* from a client entity that resides in an environment. After performing an action *encoding*, the entity produces an output signal *data* that is transmitted to a receiving entity *RcvCEFSM*. In SDL, signals between processes are transferred using signal-routes if they are contained in the same block. If the processes are contained in different blocks, signals traverse channels. After the transition, *SendCEFSM* remains in the state *wait_client_data*. Thus it can handle other *client_data* at the state. The implementation of the entity is straightforward from the state transition diagram. After the brief implementation from the diagram, the action *encoding* may be refined in detail.

RcvCEFSM is a receiving entity for the *data*. After performing the action *decoding*, it moves to the state *processing*. Note that the entity does not have any output signals, which is possible because outputs are optional in a transition.



(a) State transition diagrams

(b) Implementation in SDL

Figure 9. Simple Data Transfer Using Basic CEFSMs

Example of Predicates

Figure 10 describes a part of a call setup that reads a telephone number with nine digits. The entity *dialingCEFSM* initializes its local variable *counter* to one and generates *dialing_tone* signal to initiate a dialing tone. In the state *dialing,* it receives an event *dial* with a parameter *digit,* which means that the digit is pushed by a caller. The entity receives eight more digits at the state. After receiving the ninth digit, the machine moves to the next state *connecting*, which means the entity has the callee's phone number composed of nine digits. While moving to the state, the entity generates a signal *connect_req* to request a connection with the callee. If the CEFSM doesn't have predicates, *counter<9* and *counter=9*, the machine needs at least 10 states, i.e. *initial state* and *states to represent each digit received.* Note that when the predicate *counter<9* is true, the entity remains at the same state.

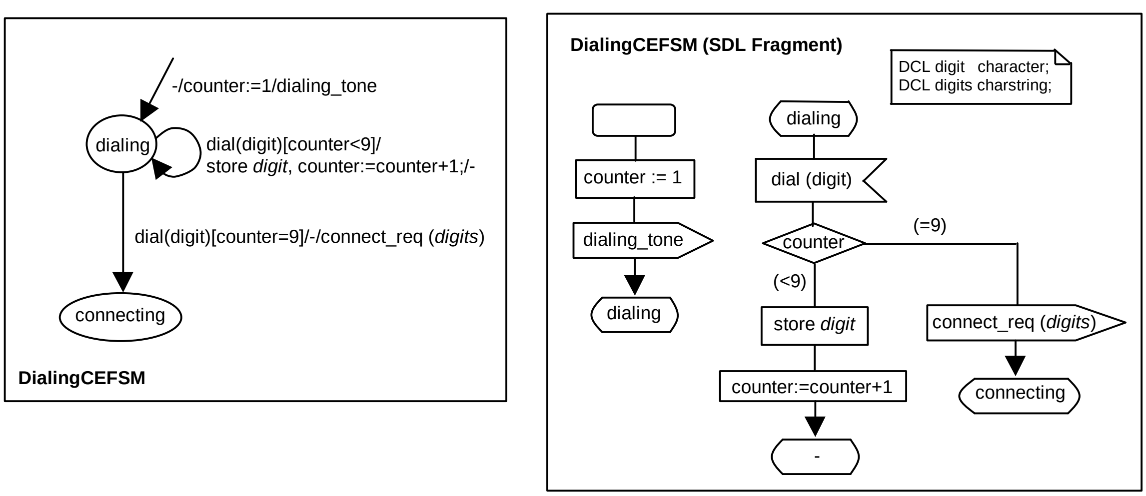


Figure 10. Nine Digit Dialing with Predicates

Example of Predicates after Actions

Figure 11 shows an error detection method using a checksum procedure *checksum()* in which if the result of the procedure, *rst*, has zero, there is no error in the received message *d*. If it is not zero, the message has an error and a notification message is sent. Note that the predicates *rst = 0* and *rst != 0* are evaluated based on the action *checksum()*. The control depends on the result of the action.

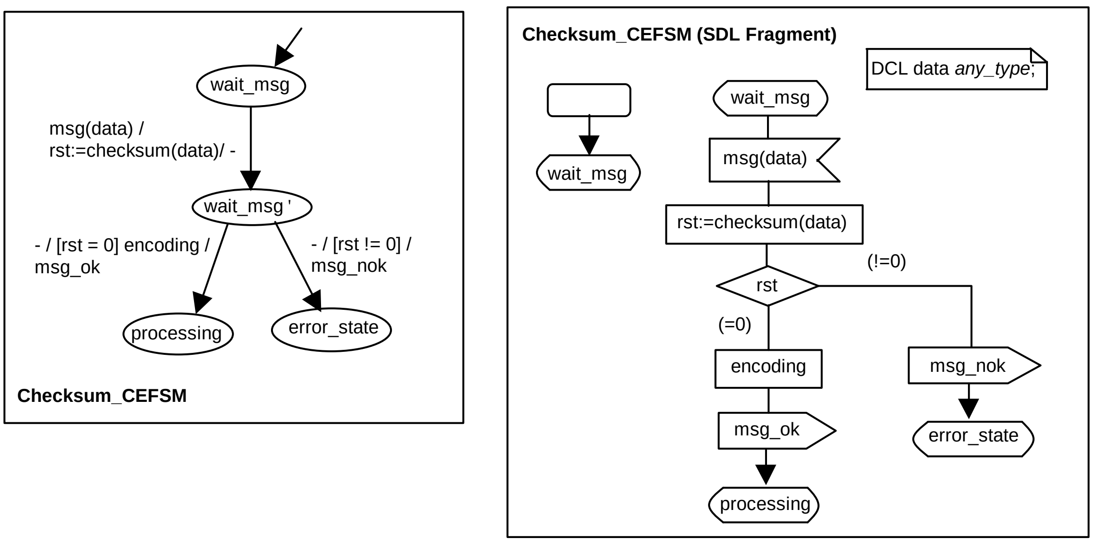


Figure 11. Checksum with Predicates after Actions

Example of Timers

Figure 12 presents a data transfer with an acknowledgement in a predefined time, 10.0. To guarantee the receiving of a message *ack*, the entity *SendAckCEFSM* sets the timer T_1 that has a default time value 10.0 when it is declared. After performing an action *encoding* and sending data by *send (date)*, the entity waits an acknowledgement message in a state *wait_ack*. If the timer T_1 expires before the event *ack*, the entity moves to the state *error_state* emitting an output signal *error*. Otherwise it resets the timer T_1 and moves to the next state *acked*.

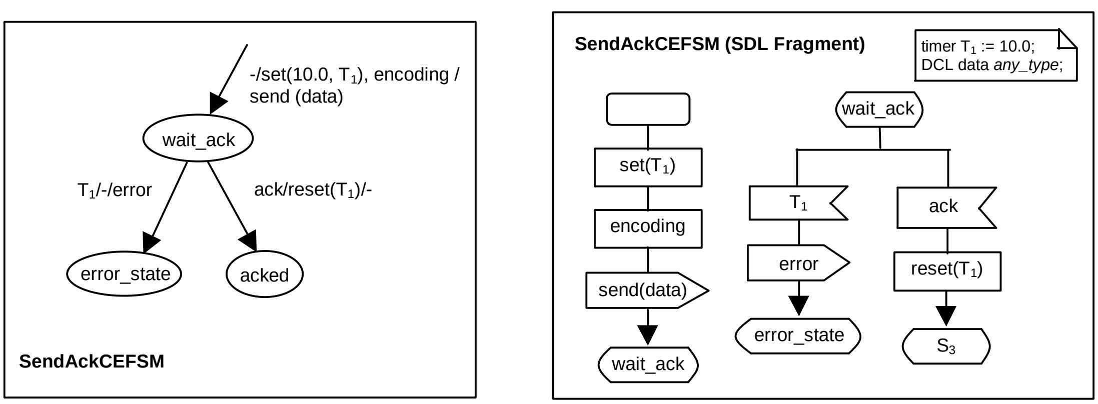


Figure 12. Example of Confirmed Data Transfer with a Timer

4. Composition of CEFSMs by Merging States

Context

In general, telecommunications systems have several entities for each system to fulfill its required functionality. The entities are composed of several states and transitions. It is necessary for designers to define an entity with an arbitrary number of states from the CEFSMs presented in the *CEFSM* pattern.

Problem

How can CEFSMs be merged to implement the behavior of a more complex entity?

Forces

Complexity of the Entity: If an entity is implemented by merging the CEFSMs that have already been developed, the size of the entity can be too large to manage the entity. As an extreme case, a system can be implemented in one huge CEFSM. But it is unreasonable in terms of development and maintenance. The designer must design the system in subsystems.

Solution

A CEFSM can be expanded by merging with other CEFSMs. There are three common ways to merge: source merge, sequential merge, and mutual merge.

Source Merge

As Figure 13 shows, if two CEFSMs have the same initial state S_1 and the different transitions for events e_1 and e_2, they can be combined to have an initial state and two transitions with each target state. This is commonly occurred when a state receives several potential input events. Note that the merged entity should do both actions a_1 and a_3 and both outputs o_1 and o_3 during the initial transition. The designer must check the effect of the merging that may cause the different result compared to the result of the original CEFSMs. For example, if a_1 was to set a variable to zero and a_3 was to set the variable to one hundred, the result of the combined entity is different from the original ones. In that case, one of the initial states should change its name to a new one, for example, S_1'. It is also possible to have conflicts between a_1 and a_4, and a_3 and a_2 at the merged entity.

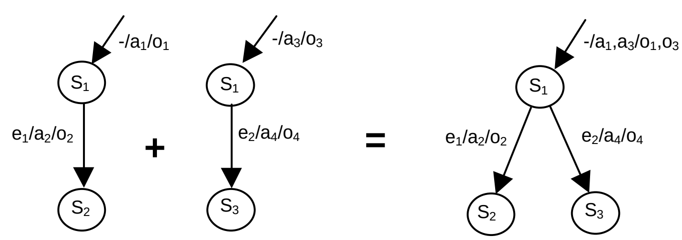


Figure 13. Source Merge

Sequential Merge

If a target state of a transition in a CEFSM is the same as a source state of a transition in another CEFSM, we can merge the target state with the source state to generate a sequential CEFSM. This situation is common when an entity processes a sequential input from other entities.

Figure 14 shows the merging in which S_2 can be reached either from state S_1 or through the initialization transition. In the combined version, the CEFSM goes to state S_2 only through the state S_1. As in *Source Merge*, the designer must check the effect of the merging that may cause different results.

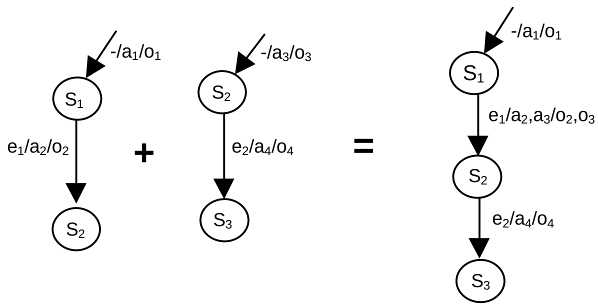


Figure 14. Sequential Merge

Mutual Merge

This is common when a machine returns its original state after a sequence of actions. The source state of one CEFSM matches with the target state of the other and vice versa. The designer must identify the initial state of the combined CEFSM. Figure 15 presents the mutual merging where S_1 is the initial state of the combined CEFSM.

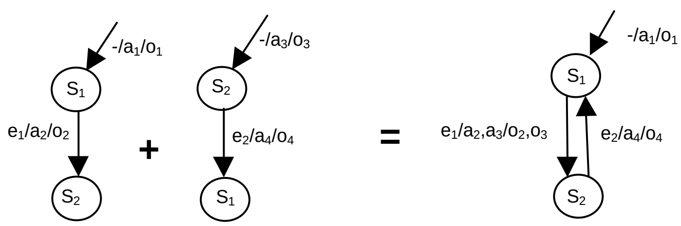


Figure 15. Mutual Merge

Translation into SDL

Source Merge

Figure 16 shows the implementation of *Source Merge* in SDL. It needs two transitions from the state S_1. As mentioned before, the events, e_1 and e_2, should be distinct.

Sequential Merge

As Figure 17 illustrates, there are two transitions where the target state of the first transition becomes the source state of the second transition.

Mutual Merge

Once the initial state is identified, the implementation is straightforward. In Figure 18, the state S_1 is the initial state of the combined CEFSM.

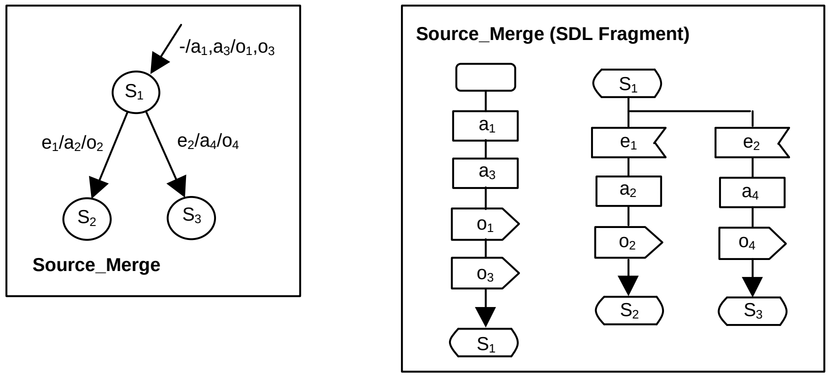


Figure 16. Source Merge in SDL

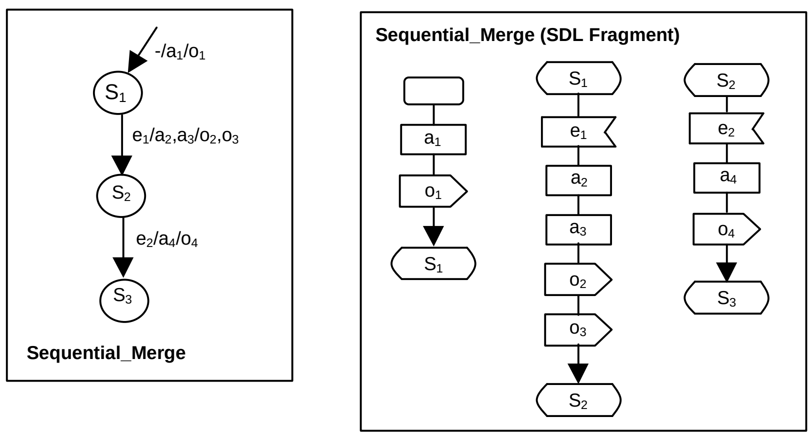


Figure 17. Sequential Merge in SDL

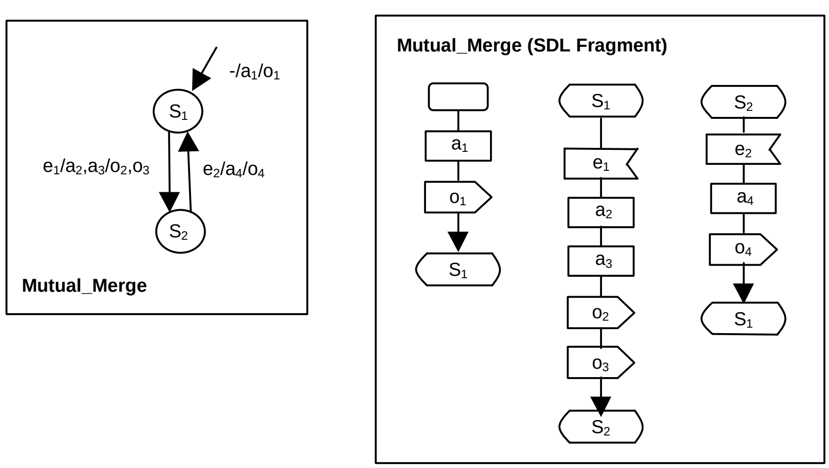


Figure 18. Mutual Merge in SDL

Example

Source Merge

Suppose we are designing a system to handle several packets. We can consider two cases: One is to handle a connection request packet *c_req* and another is to handle a disconnection request packet *d_req*. At Case 1, we perform a preparation action for the connection request *c_prepare* and emit a ready signal *c_prepare_ok* during the initial transition. At Case 2, we perform *d_prepare* and *d_prepare_ok* during the initial transition. If the two cases are combined, we will have Case 3 where the entity performs the preparation actions and outputs for the both cases during initialization. We do not describe the implementation in SDL because it is trivial.

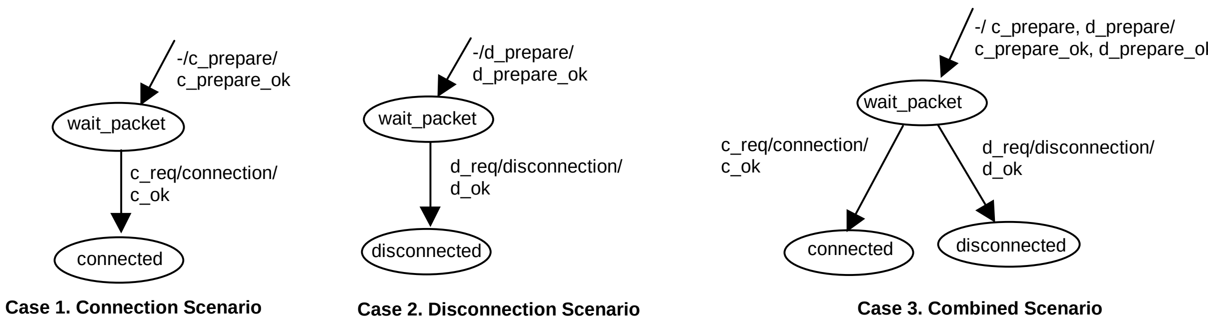


Figure 19. Packet Handling CEFSM using Source Merge

Sequential Merge

In this example, let's assume that the system can be disconnected only after the connection setup is finished. Case 1 is the same as the *Source Merge* example, but in Case 2, the source state should be in the *connected* state because it is a prerequisite for the disconnection. In the combined case, the state *disconnected* can be reached through the state *connected.*

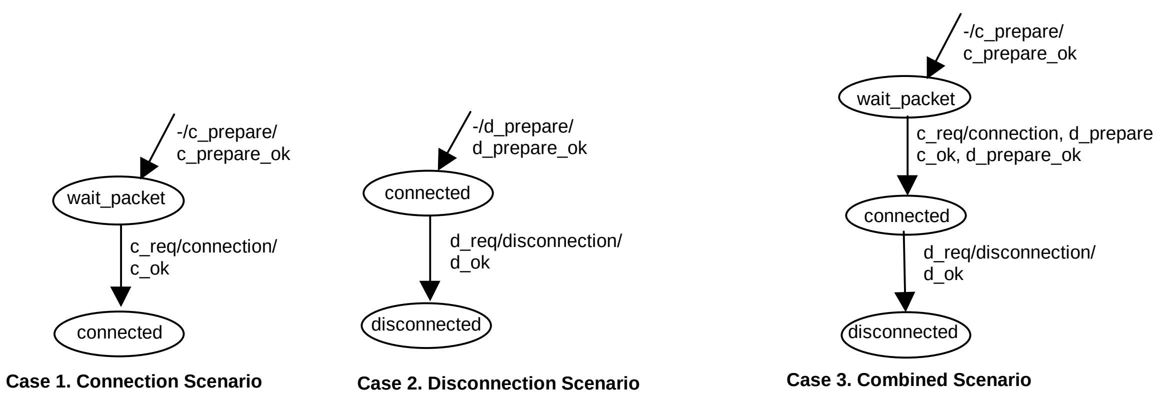
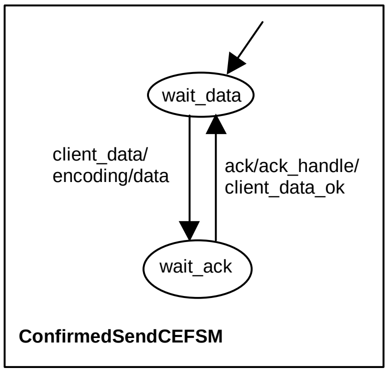


Figure 20. Packet Handling CEFSM using Sequential Merge

Mutual Merge

This example shows a confirmed version of the simple data transfer of *SendCEFSM* described at the *CEFSM* pattern. Upon receiving the event *client_data* from outside client, *ConfirmedSendCEFSM* performs *encoding* action and outputs an event *data* to send data to its receiving peer. The entity moves to the state *wait_ack* in order to handle the event *ack* from the received entity. If the entity receives the event, it notifies the client of the receiving of client data and returns to its original state so that it can transfer other *client_data*.



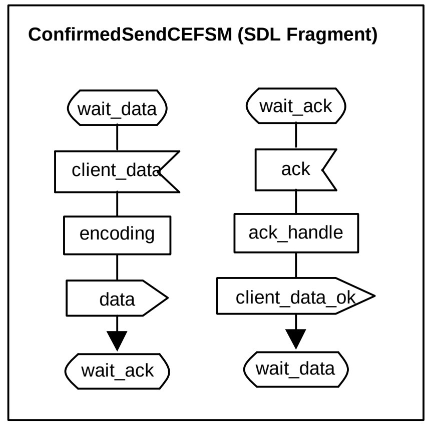


Figure 21. Confirmed Data Transfer using Mutual Merge

5. Handling Repeated Events with Timing Constraints

Context

In telecommunications systems, one event is generally enough to initiate a reaction for the event. But in some cases, the systems need a set of events to start actions. In other words, a CEFSM collects information from several events in a given time, and then it performs actions based on the information. For example, if an entity wants to encode a packet that is composed of several fragments and each fragment is given by one event from a sending entity, the entity must receive all fragments at first. In that case, the entity starts the encoding after the events for all fragments. It is reasonable to impose timing constraints for the events to avoid unlimited waiting.

Problem

How can we handle several events with timing constraints?

Forces

Timing Constraints: We can have two kinds of timing constraints in repeated events, one is a timing constraint for each event and another is the total time limit of all events. The real values of the constraints are dependent on specification.

Solution

We can merge the entity *Repeat_Predicates* and the entity *CEFSM_Timers* of Figure 4 to make a higher level pattern. The entity *Repeat_Predicates* uses the predicates *counter* < *MAX* and *counter* = *MAX* for the repetition. Figure 22 shows the merging in a state transition diagram. Note that we rename the states, actions, and outputs in the merged diagram. Initially, a timer T_1 is set for the timing limit of each event *e*, and T_2 is set for the timing limit of all events. An integer variable *counter* is set to one in order to count the number of events that have happened. At the state S_1, the entity handles the event *e* until it receives all the events, for example, *MAX*. If the *counter* is less than *MAX*, the entity increases it and sets the timer T_1 again. Setting a timer always performs *reset*() implicitly and the corresponding timer signal is removed from the input queue, if it exists [Ellsberger+97]. If the timer T_1 or T_2 is expired, the machine moves to a state to handle the error case.

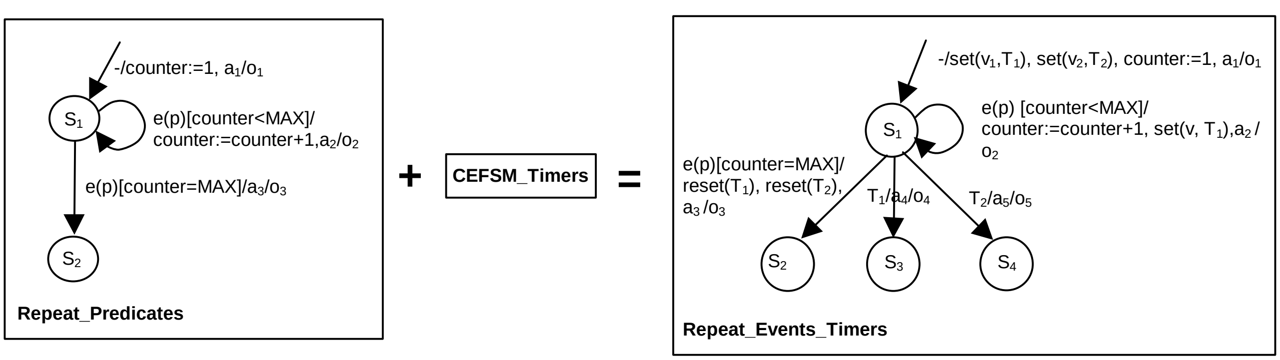


Figure 22. Handling Repeated Events with Timers

Translation into SDL

The translation is obtained easily from the state transition diagram.

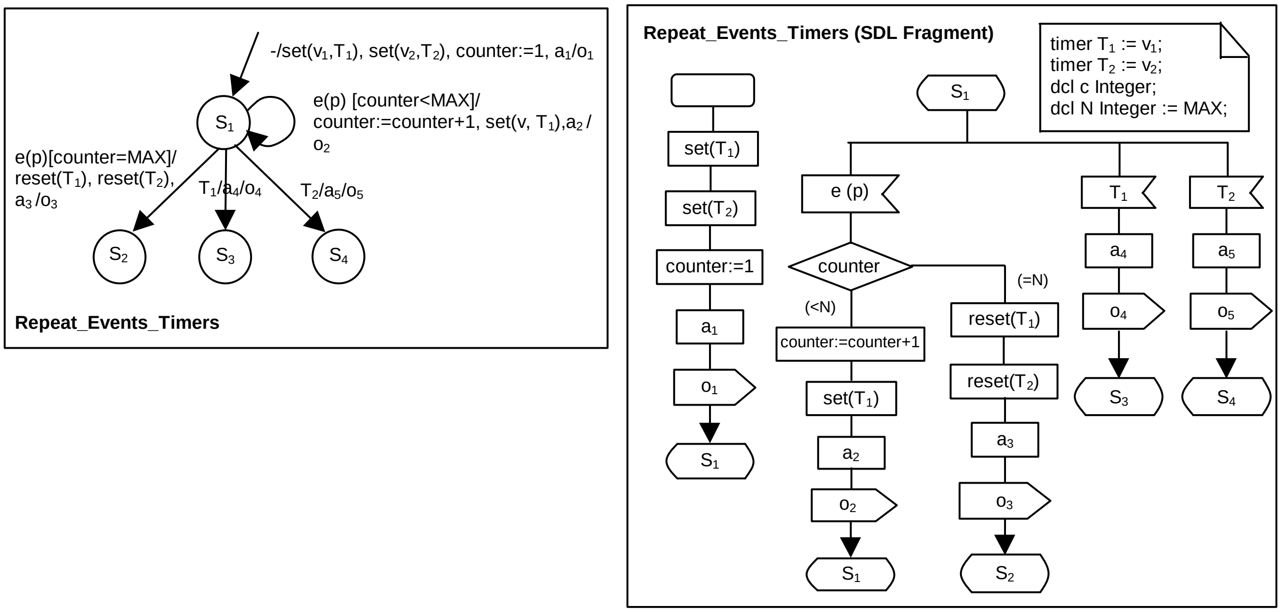


Figure 23. Translation of Repeated Events with Timers

Example

This example is a confirmed version of the nine digit dialing of the example of predicates. If a caller does not push a digit in a given time bound, the entity notifies the caller by giving a special beep to redial the digits. This situation is implemented at the transition from the state *dialing* to the state *error_state*. Note that the example only checks the timing constraint for each event. The caller, therefore, must dial a digit in a time limit T_1 but there is no

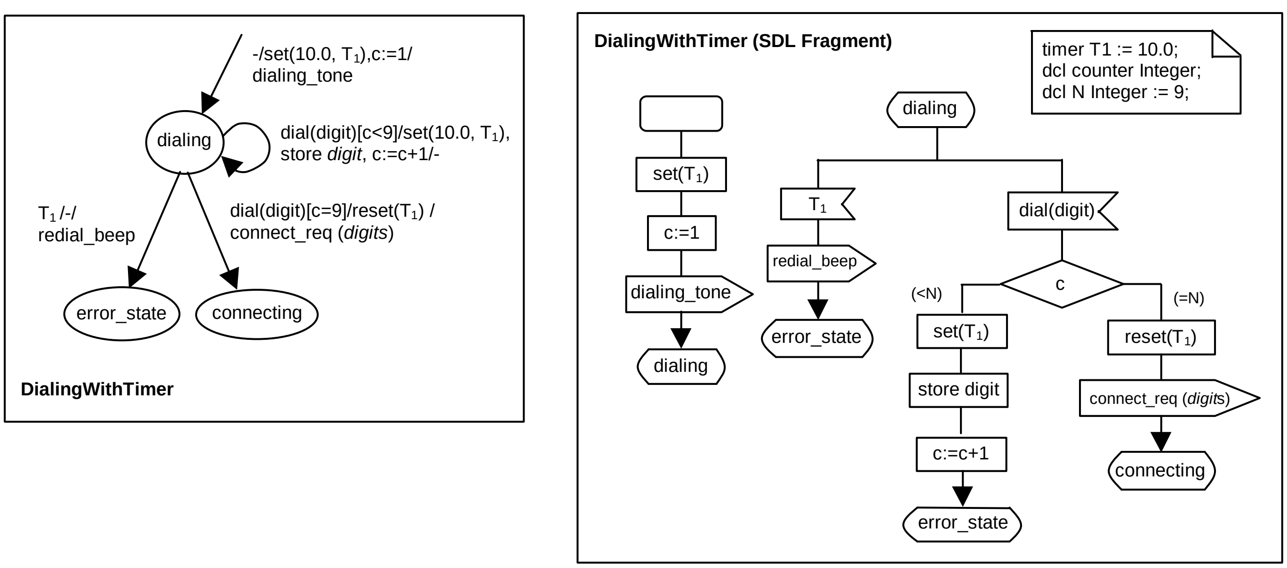


Figure 24. Nine Digit Dialing with a Timing Constraint

constraint for the whole dialing.

See also

The *TimerControlledRepeat* pattern [Geppert+96] repeats a message sending to avoid message loss during data transfer. If a sender entity does not receive an expected acknowledgement in the given expiration time from a receiver entity, the message is repeated by the sender. This pattern, however, does not consider the number of repeats. Thus the entity does not know how many times it would repeat the action, although it defines a maximum number of repetition.

6. Dynamic Handlers

Context

In a client-server environment, there may exist several requests from clients at the same time. The server must handle the requests concurrently to achieve high performance.

Problem

How can a server handle the multiple requests dynamically?

Forces

Server Capacity: If a server handles many clients at the same time, its performance will be degraded. In order to provide services properly, the server needs to find a tradeoff between performance and maximum number of clients that the server can handle at the same time. Since the performance depends on the server's capacity and types of requests from clients, it is not always possible to find an optimal solution.

Static and Dynamic Approaches: We can consider two types of server in handling the multiple clients. One is a dynamic approach where a server creates a new handler to handle each client. This approach lets the server maintain exactly the number of handlers it needs. Another is a static approach in which the server provides a finite number of handlers from the very beginning to reduce the overhead of handler creation. Since each approach has advantages and disadvantages in a specific situation, we cannot say which one is better than another. The solution presented in this pattern shows the dynamic approach. The static solution can be made in another pattern.

Solution

As Figure 25 (a) shows, an entity *Admin* creates an entity *Handler* dynamically for each client [Ellsberger+97]. The server block *Admin-Handlers* is composed of one static *Admin* and several dynamic *Handler*s. The *Admin* entity that is created at the system initialization time waits for connection request signals from clients. Upon receiving a signal *connect_req*, the *Admin* entity creates a *Handler* entity giving the address of the client, *client_id*. The address is obtained by either a parameter of the signal *connect_req* or analyzing the origin of the signal. The dotted line from *Admin* to *Handler* indicates the creation of a new entity. The *Handler* entity sends the signal *connect_ok* to the corresponding client that requested the connection. When the client receives the signal *connect_ok*, it must also know the address of the *Handler* for the later communications. After the connection setup, both the *Handler* and the client perform communications through *request* and *reply* signals. The *Admin* entity can handle other *connect_req* signals from different clients while the *Handler* entity is serving the client. When a session ends, either *Handler* or client can initiate the disconnection by sending a signal *disconnect_req*. In the figure, client initiates the disconnection. The *Handler* entity informs the *Admin* entity of the termination of the session by sending a signal *terminate* and ceases to exist.

The diagram (b) of Figure 25 describes the signal flows among the entities. The diagram (c) shows the state transition diagrams for the *Admin* and a *Handler*, in which the process termination, denoted by a cross symbol, represents the end of process. Actually the cross symbol is not a standard notation of CEFSM. When a session starts and ends, the *Admin* entity keeps the information on a *Handler* entity by the actions *allocate* and *de-allocate*. These actions might be refined in later developments.

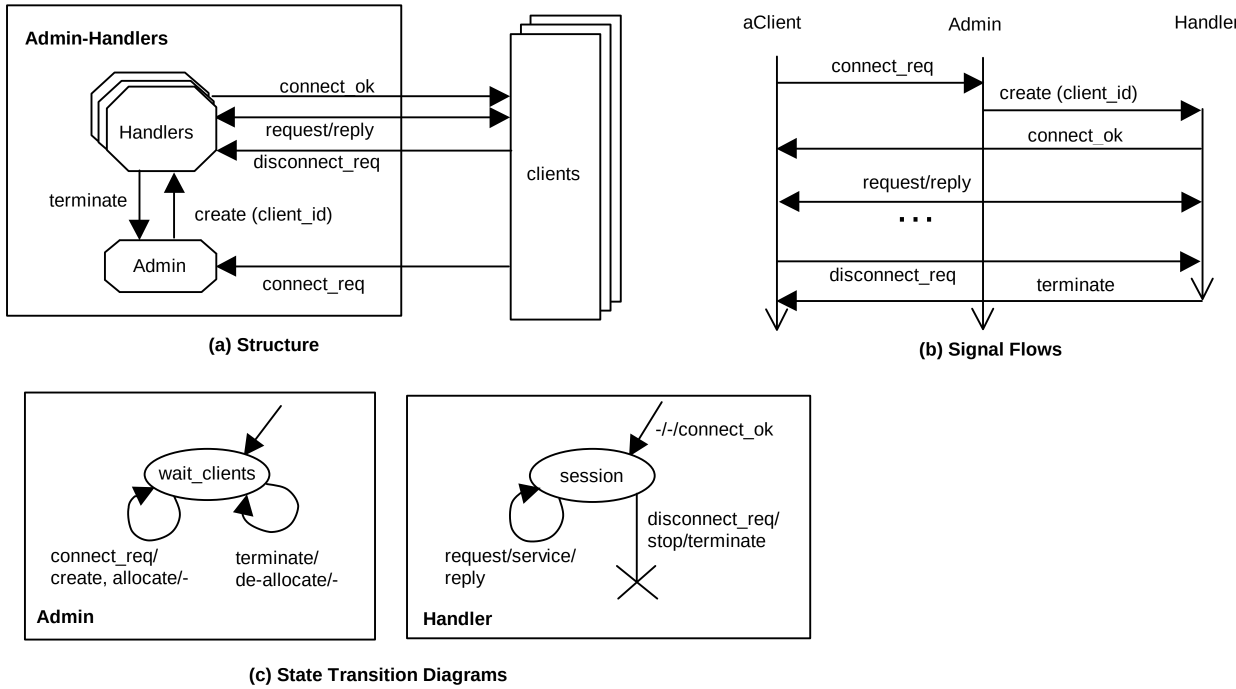


Figure 25. Dynamic Handlers for Multiple Clients

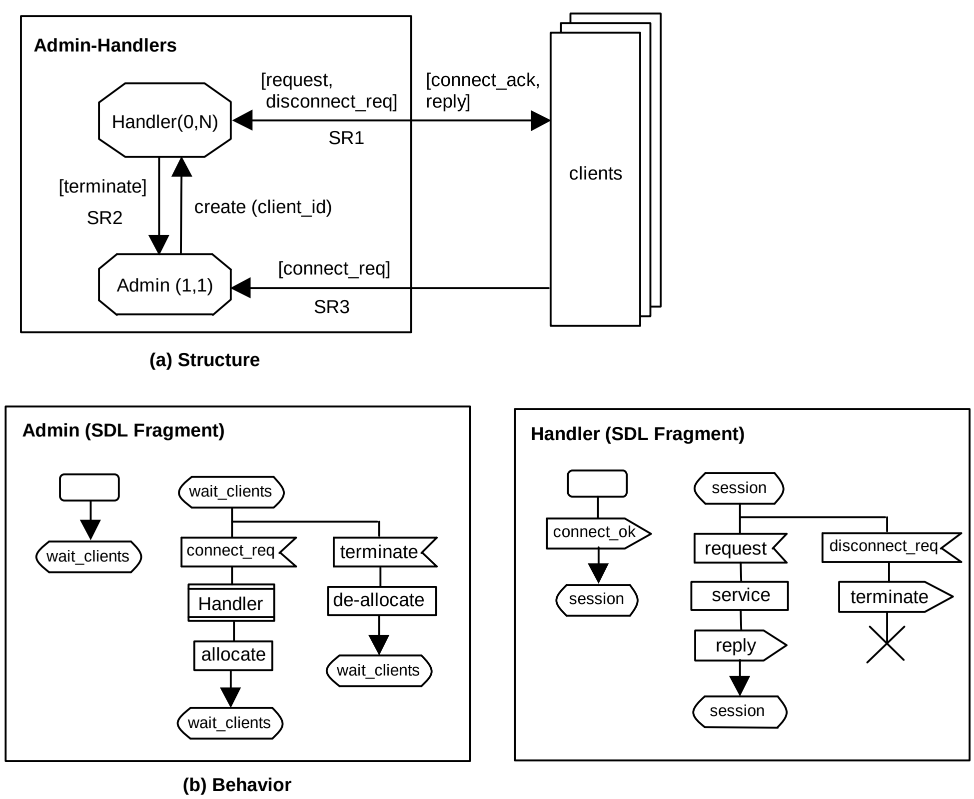


Figure 26. Translation of Dynamic Handlers into SDL

Translation into SDL

Figure 26 (a) shows the translation of the structure of *Admin-Handlers* block. In SDL, the static structure of a system is described by a hierarchy of blocks [Ellsberger+97]. A block can contain other blocks, resulting in a tree structure. The leaf block is made up of one or more processes describing its behavior. We have two processes, *Admin* and *Handler*, with the initial and maximum number of instances. Processes are connected with each other and to the boundary of the block by signal routes, for example, *SR1*, *SR2*, and *SR3* in the diagram.

The translation of the behavior is similar to other CEFSMs except the creation and termination of a process. Although the diagram does not describe it, we can include a verification step for the process creation. In case of successful creation, the *offspring* value of *Admin* entity has the instance number of the created entity. If not, it has the value *null*.

Example

Figure 27 shows a simplified call handling of a switching system. A server block *Call_Handling* is composed of an entity *Call_Admin* and an entity *Call_Handler*. When a caller tries to make a call, a signal *off-hook* goes to the entity *Call_Admin* to indicate a call trial. The entity creates a *Call_Handler* to assign the call. The entity *Call_Handler* sends a signal *dialing_tone* to indicate that it is ready to receive dialing digits, which is composed by merging the entity *DialingWithTimer* presented in Figure 24 with a CEFSM that has *connecting* and *talking* states.

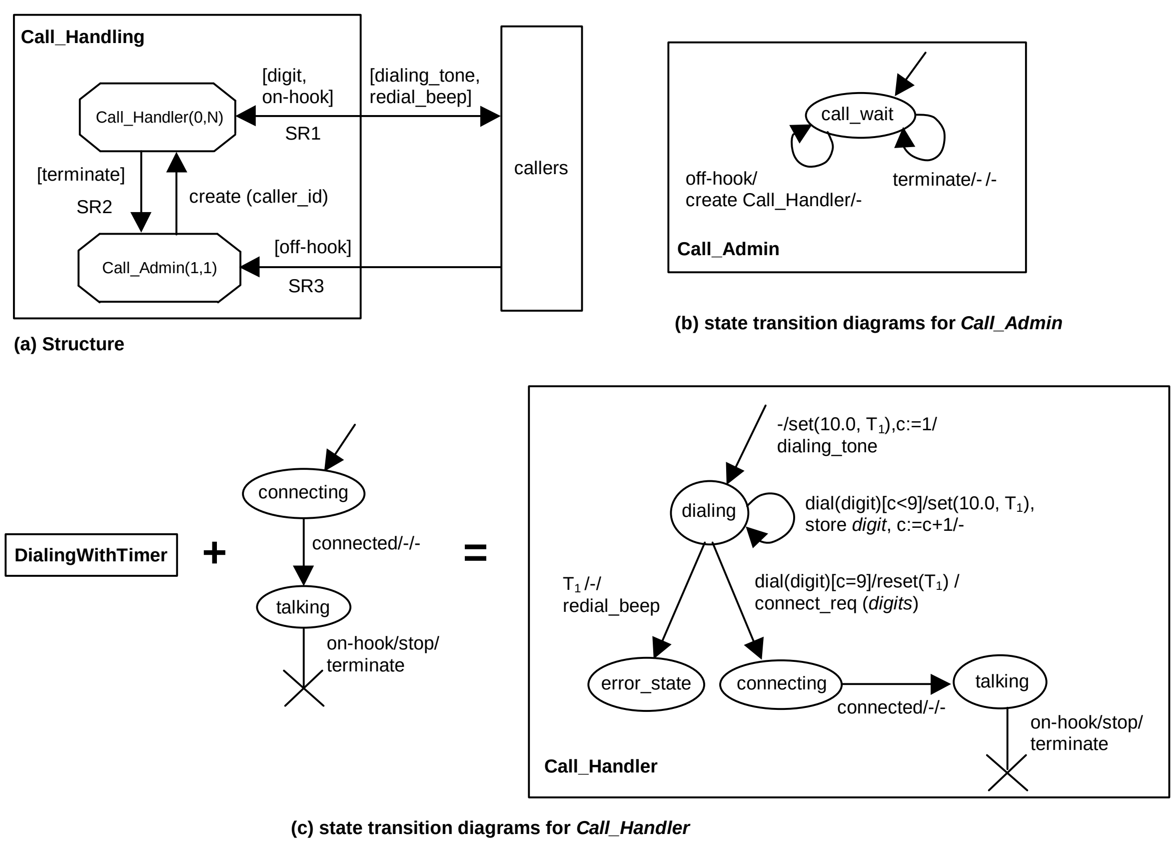


Figure 27. Call Handling with Dynamic Handlers

See also

This pattern is similar to a pattern *DynamicEntitySet* [Geppert+96] where *EntityAdministrator* creates and forwards requests to a *TerminatingEntity*. The difference is that our *Admin* entity does not forward the requests after the creation of a *Handler* entity where communication is performed directly between the *Handler* entity and a client.

7. Conclusion

In this paper, we introduced patterns describing the use of communicating extended finite state machines. A CEFSM represented in a state transition diagram is easy to understand because it describes the states, the events, and the actions clearly. It can also describe the behavior precisely and formally. The formal description can provide several advantages, for instance, in the tool support for the patterns and the design evaluation of the initial design made by the patterns. We also showed the implementation of the patterns in a specific language SDL. We expect that this specific implementation will be helpful with potential to be further reused in a real application.

In other research, [Geppert+96] proposed SDL patterns that represent their solutions and implementations in terms of SDL, whereas we first model our solution in CEFSM, and then we translate it into SDL. As a result, the modeling in CEFSM can be applied to other formal description techniques such as Estelle, and so on. [Yacoub+00] presents a pattern language of finite state machines that provides solutions for the finite state machines in terms of an object-oriented design. It proposed a basic FSM pattern and its extensions handling several design issues such as state transition mechanisms, design structure, state instantiations, exposure of internal state, and the machine type. In our paper, we focused on the state transition diagram and the implementation of a diagram in SDL.

The patterns presented in this paper describe relatively low-level behavior. In future work, we will add high level patterns obtained by composition of the primitive patterns such as the pattern *Handling Repeated Events with Timing Constraints*. Also, our patterns will be supplemented with the architectural patterns that will describe the structural aspects of telecommunications systems.

8. Acknowledgements

We thank Michael M. Wu for his helpful and insightful comments during the PLoP shepherding process. We also thank Electronics and Telecommunications Research Institute for their financial support.

References

[Buschmann+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume1: A System of Patterns*, John Wiley & Son Ltd, 1996.

[Ellsberger+97] J. Ellsberger, D. Hogrefe, and A. Sarma, *SDL – Formal Object-Oriented Language for Communicating Systems*, Prentice Hall, 1997.

[Gamma+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Geppert+96] B. Geppert, and F. Rößler, *Pattern-based Configuring of a Customized Resource Reservation Protocol with SDL*, SFB 501 Report 19/96, Computer Science Department, University of Kaiserslautern, Germany, 1996.

[ITU-T96] ITU-T, *Recommendation Z.100: Specification and Description Language (SDL)*, ITU-T, Geneva, 1996.

[Rozenblat01] Boris Rozenblat, *CS475 Telecommunications Software Engineering*, http://goanna.cs.rmit.edu.au/~borisr/CS475.html (current 10 May, 2001)

[Yacoub+00] S. Yacoub and H. Ammar, *Finite State Machine Patterns*, Pattern Languages of Program Design 4, pp. 413 – 440, Addison-Wesley Longman, 2000.