The Key FSM

A finite state machine for better system development

Arie Avnur
AA Software Development
AA-SW-DEV.COM

I. Introduction

This paper introduces the *Key FSM* (KFSM) – an extension of UML's Statechart with additional features for effective system development.

A. FSM's Power and utility

The FSM model is very popular among software and hardware developers, a fact that testifies to its usefulness. Although somewhat philosophical, it is beneficial to discuss and try to understand the reasons for this power. The following discussion in this subsection represents the author's opinion.

The FSM's main power is in the notion of ***state***: It is a well-defined, easy to communicate and document condition of the entity whose behavior is modeled by the FSM. E.g. a person may be described as being in the "sitting", "standing" or "running" state; a car in "parked", "traveling forwards"," backing up" or "idling" state. Each state defines a static behavior of the element modeled by the FSM. That behavior of a well-defined state should not change. The modeled element's behavior changes when it transitions to a different state, in response to an event. A sequence of transitions in response to events represents the element's dynamic behavior. This "divide and conquer" approach is how the FSM enables system analysts and implementers to better manage complex dynamic behavior.

The state is therefore very important to both model and implementation clarity and precision. This understanding leads us to put more emphasis on the state's conditions and actions and to prefer the Moore FSM model (see below). According to this approach, transitions should be used only as means to move to another state in response to events. ""Light" actions may be specified on transitions, like logging or posting a message to a user—if it does not require confirmation; waiting for a user response is a lengthy process and it justifies an intermediate state.

The FSM model and its implementation can and should play a role in constructing dependable systems that recover gracefully and orderly from improbable system failures. KFSM and the design and analysis tools it provides are geared to support this novel engineering aim.

B. Traditional FSM Models

1. The Basic FSM Model

The basic FSM defines a set of states, one of those states as an initial state, a set of events it responds to, a set transitions that are a mapping function: state X Event → (next) state and a subset of the states that are defined as terminal states – where the operation or response of the states-machine terminates. A mathematical definition of the classical FSM see the sidebar.

2. The Mealy Model

This model defines actions or output for the transitions and not the states. These output actions are required then to bring the system from the transition's source state to its "next" state, and could be different for each transition into the same state. This arrangement divides the responsibility of keeping a state in compliance with its requirements between several transition actions, increasing complexity and opportunities to err.

3. The Moore Model

Moore defined an FSM model that determines outputs as a function of the state. As stated in the introduction, the notion of the FSM as a model for dynamic behavior relies on a solid definition of static behavior in each state, an approach that favors the Moore model.

Practically, some actions or output might be required that are associated with transition; those however should be secondary, like logging, posting a message (not waiting for a confirmation) and similar short processing operations that keep the transition as short or quick as possible.

FSM's Definition

(Follows [1], [10] and many other sources, symbols changed.)

$M = (S, E, D, T, L, s_0)$; where:

- S is the set {s} of states,
- E is the set {e} of trigger events or input that cause transitions
- D is the set {d} of the FSM's output events or actions
- T is the transition function that maps QXE to Q. $T(s_i,e)$ is the next state s_j of the machine.
- L is the output function that maps SXE to S. $d_i = L(s_i,e)$ is the set of output events generated by the FSM when in the state s_i for all e_j (Moore model) or when transition $T(s_i,e_j)$ happens (Mealy).
- s_0 is the initial state of the FSM

4. The Statechart Model

Fast forward quite a few years; UML 2 adopted the Statechart FSM model which is an extension of the basic FSM model. It adds the following features:

(1) Orthogonal parallel FSMs or "Zones" per the UML notation.
Several independent FSM's can be operating in parallel. Events are transferred to all of them in parallel.

(2) FSM hierarchy
A state may encompass a group of parallel FSM's.

(3) History restart
A sub FSM contained within a higher level state may have a "history" attribute that causes it to return to the last state it was in when the containing state was exited, instead of starting every time from the initial state.

A more complete description and specifications of Statechart can be found in [4].

II. Key FSM Concepts and Features

In this chapter we will describe and discuss systems and software engineering concepts that can be added to the common FSM reviewed above.

A. Control Structure

An FSM defines its output values for each state, weather that output is set at the state (Moore model) or by the transitions into the state (Mealy model). Can any abstract variable be an output? How does it relate to system variables and the FSM scope? Should there be a relation between the outputs of various states? KFSM addresses those points with the *control structure* concept; at an abstract level, it is a set of data elements (i.e. application entities or their properties) defined as controlled by the FSM. The control structure must be within the scope of the process that run's the FSM so it can control them. Additionally, all states must define values of all control structure's elements. For good system engineering, if some of control structure's elements are of no consequence at some states, they can be defined as "N/A," and the reason for that choice documented as part of the KFSM design documentation. The control structure serves then to guide the analyst or designer to define all of the FSM output, improving consistency and reducing the risk of omission error.

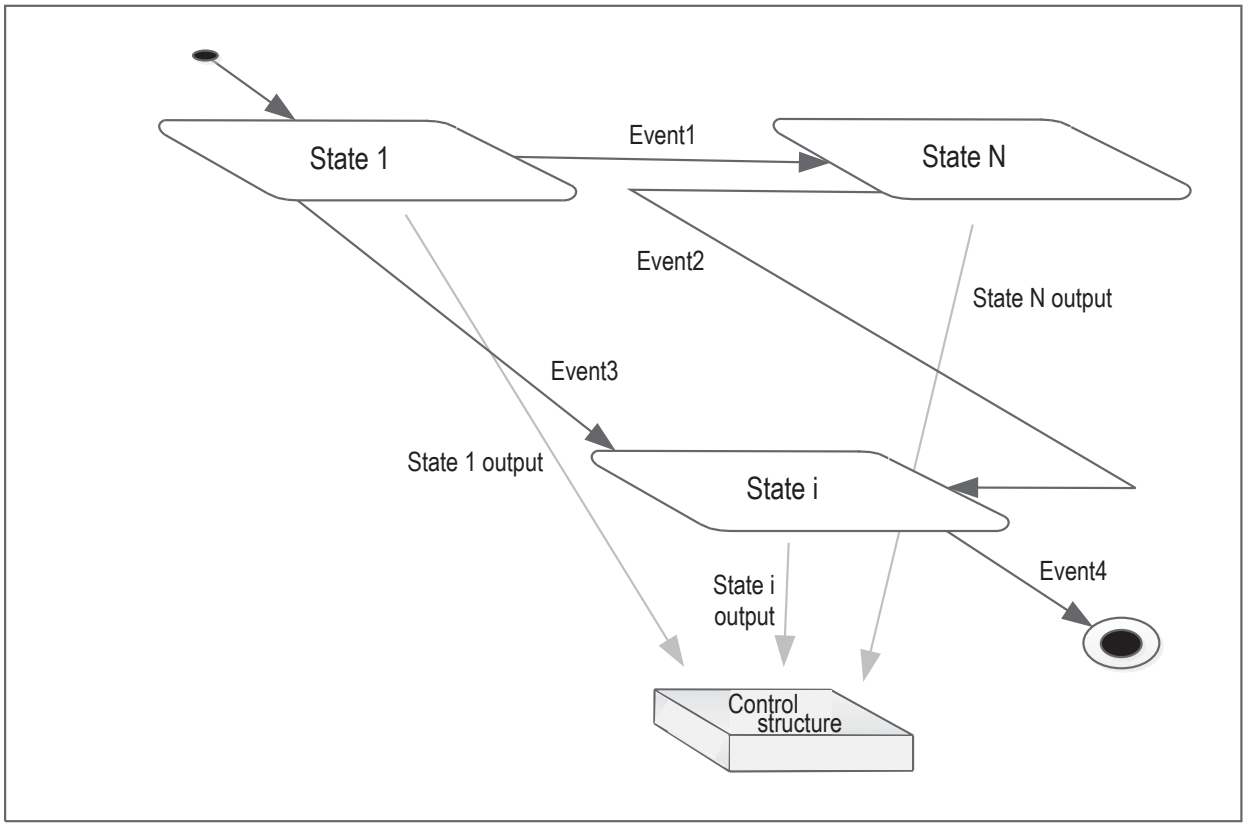


Figure II-1: Control Structure as the output of all states

A control structure case Key FSM can treats as special is when it—the control structure comprises variables of a primitive type only that can be set directly; we will identify such a control structure as a *control variables structure* or shortly *control variables*. In that case the Key FSM state definition can include numeric values of the control variables and the KFSM implementation sets those values from the definition, there is no need for a line of code for implementation as will be seen later.

An example FSM is used to throughout this work—a cruise control system control FSM. See Appendix 1: . It specifies a control structure comprising:

- *ControlOn – A Boolean property indicating that the system is engaged and controls the car's speed.*
- *setPoint – a continuous property (real or double variable) representing the set speed.*
- *accelerating – a Boolean property indicating that the set point is to be increased at a constant rate.*

These properties/variables in turn control other parts of the system, like a control and acceleration tasks. Note that all the example's FSM states specify the values of these three control structure elements.

B. Key FSM Models for Various Development Phases

Different development phases produce different artifact types: requirements and design specifications, code and lastly, test cases and results. Analysts and developers may need to use specialized models and tools for each phase; Key FSM offers another option.

Development is done in steps that deal with various aspects of the developed object: requirements define conditions, constraints and output, design outlines how the former are to be materialized, and implementation produces the operational code that is the product. All development processes include those steps in one form or another, even if not documented and done mentally. Finite state machines are used for those development phases routinely. Analysts and requirements engineer use FSM models to specify dynamic behavior of requirements. Architects and developers use FSM models to specify implementation, and eventually materialize it. But they all use fundamentally the same model structure. Key FSM offers a different solution—an FSM part dedicated to the special needs of each of those development steps and relations between them.

The key to this feature is the Key FSM state that holds distinct parts for the different development steps and thus forms a layered state definition and a layered FSM as Figure II-2 depicts.

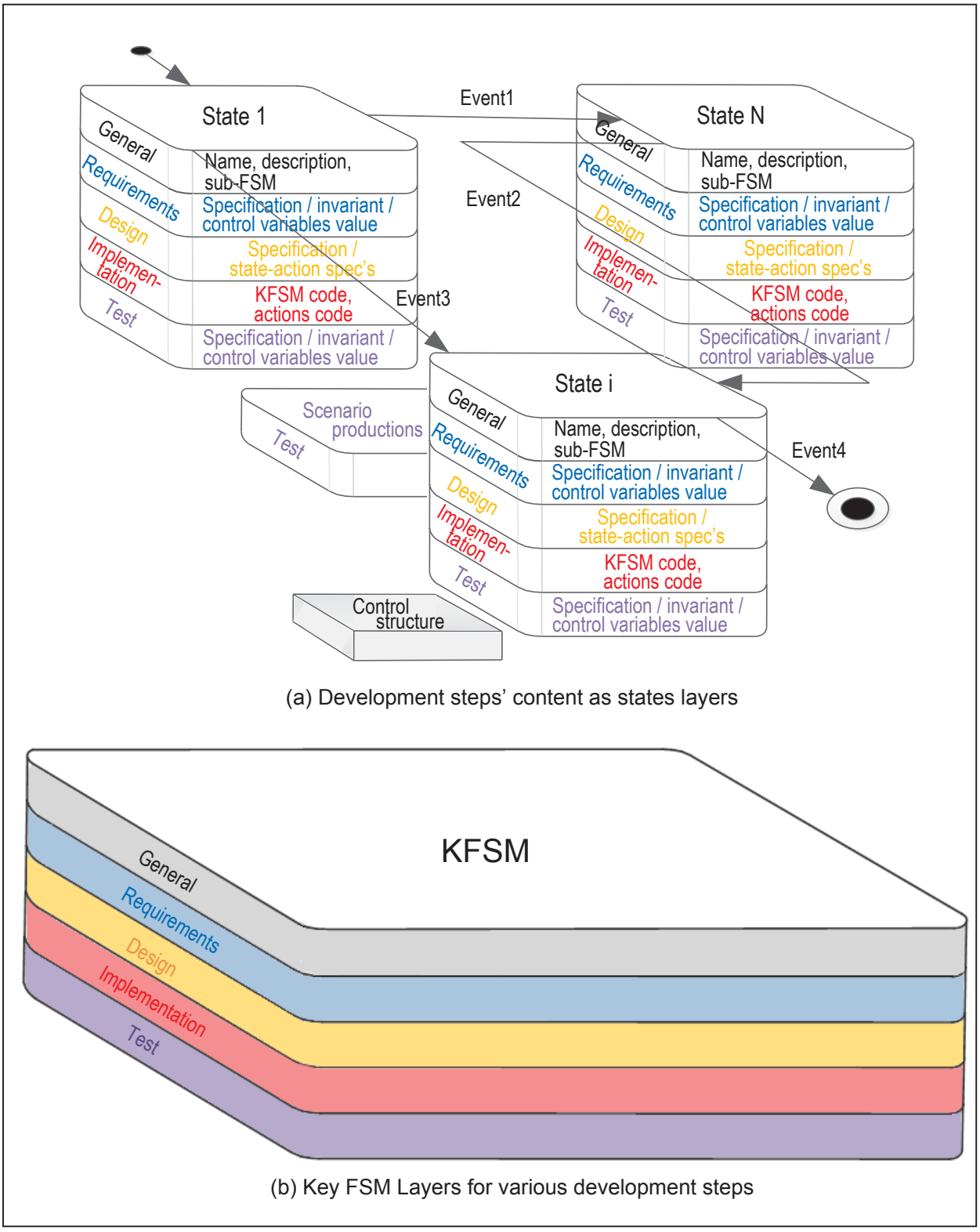


Figure II-2: KFSM layers

The KFSM state holds a layer for each of the development (see Figure II-2 (a)):

- General: the state's name, attributes, and a reference to another KFSM/s if it is specified as having sub KFSM or even multiple KFSMs as separate UML regions.
- Requirements: specifies **what** condition is the KFSM control structure required to meet.
- Design: specifies how the required conditions will be met.
- Implementation: the KFSM implementation code.
- Test: Input and output information to be embedded in test scenario.

Each of the state layers, combined with the general layer, the transitions definition and the control structure, can be viewed as a whole KFSM layer (see Figure II-2 (b)).

Following is a detailed description of those layers.

1. General layer

The KFSM general layer represents the KFSM's basic structure and includes:

(a) states' corresponding general layer properties:
   a. State's name,
   b. Description,
   c. State's attributes: terminal, timeout,
   d. Sub-FSM reference, for states defined by a whole FSM or even multi-zone concurrent FSM's (UML),

(b) KFSM's transitions – all the mapping of events to transitions, as defined by the state transitions table. These transitions are based on the KFSM *transitions model* – the combination of all the transitions related features that gives KFSM its capability to address design and run time challenges. See section II.E.

The general layer is actually an FSM in its common form, as used quite often by analysts and developers. The other layers add development phase specific information. Table ___ shows the format of KFSM's general layer state definition table:

Table II-1: General layer's states definition table

State's name	Description	Attributes
[Name]	[Description]	Terminal=T/F, timeout=T/none, sub-KFSMs: KFSM1, KFSM2… / none

And the general layer states definition table for the Cruise Control example will be:

Table II-2: Cruise Control examples states definition table

State (name)	Description	Attributes
Constant speed	The system controls the car's speed to match the set-point as close as possible.	Terminal=F Timeout=none

		Sub-KFSM=none
Accelerating	The system controls the car's speed to match the set-point while increasing the set point at a constant acceleration.	Terminal=F Timeout=Toa (factory configured) Sub-KFSM=none
Decelerating	The system does not control the car's speed nor its throttle, letting it to coast, decelerating naturally.	Terminal=F Timeout=none Sub-KFSM=none
No control	The system does not control the car speed.	Terminal=F Timeout=none Sub-KFSM=none

2. Requirements Layer

When a KFSM model is used for requirements, its states should specify what should/shall system behavior or output be in the state, or what we will call here *state requirements*. Those requirements may take one of the following notations:

(a) Requirement statement/s—the common form of requirements as used by the project or organization, e.g. the EARS notation [6], or

(b) State invariant—a logic expression over control structure entities and variables that has to evaluate to true while the system is in the state,

(c) Control variables value constraints, for a control structure comprised of control variables only. At requirements level, control variables are specified in an abstract form, like entities and their properties in a UML class diagram and to be valid, this form requires that those abstract properties are mapped to primitive or simple variables in the design. The benefit of using this form is that KFSM can implement the specified FSM with from this layer's definition, with very little coding if at all (See IV.C below).

This layer adds more formal and precise requirements information to the general layer, improving requirement model's quality. Table ___ shows the format of KFSM's general layer state definition table:

Table II-3: Requirements layer's states definition table

State's name	Description	Attributes	Requirements
[Name]	[Description]	Terminal=T/F, timeout=T/none, sub-KFSMs: KFSM1, KFSM2… / none	[Statement] or [State invariant] or [Control variable values]

And the general layer states definition table for the Cruise Control example will be:

Table II-4: Cruise Control examples requirements states definition table

State (name)	Description	Attributes	Requirements
Constant speed	The system controls the car's speed to match the set-point as close as possible.	Terminal=F Timeout=none Sub-KFSM=none	controlOn=T setPoint=V_0 accelerating=F
Accelerating	The system controls the car's speed to match the set-point while increasing the set point at a constant acceleration.	Terminal=F Timeout=Toa (factory configured) Sub-KFSM=none	controlOn=T setPoint=V_0 + DV (1 + t/Dt) accelerating=F
Decelerating	The system does not control the car's speed nor its throttle, letting it to coast, decelerating naturally.	Terminal=F Timeout=none Sub-KFSM=none	controlOn=F setPoint=V_C (current speed) accelerating=F
No control	The system does not control the car speed.	Terminal=F Timeout=none Sub-KFSM=none	controlOn=F setPoint=V_C (current speed) accelerating=F

Not: from this table we see clearly that Decelerating and No control states are functionally identical and could be reduced or combined to one.

3. Design Layer

The next step is to design **how** to implement the state requirements. KFSM's design layer offers two options for the design layer:

(a) Design specifications—textual definition of what system components do at this state, data exchanged between them, and how all this implements the state's requirements,
(b) State actions—specify entry, do and exit actions.
(c) Control variable values—if not defined at the requirements level.

The first option is straight forward—the design layer contains textual or model based specifications. The second option is more interesting since it is specific to a (UML) FSM model. Three types of actions are defined for a state: entry-performed once on the entry to the state, do-performed periodically while in the state, and exit-performed once on the exit from the state. If choosing to use this state actions model, the design layer specifies the functionality of those actions required to meet the state's requirements. Table II-5 summarizes roles and requirements of those state actions.

Table II-5: Requirements outline for state actions

	State action constraints		
State requirements notation used	**Entry Action**	**Do Action**	**Exit Action**
Specification statements	Prepare control structure entities and variables – bring them to required initial values;	Perform any repeating action required to maintain compliance with state's requirements	Clean-up
Control variable values	Set values	-	-
State invariant	Bring control structure to comply with state invariant	Maintain compliance with state invariant	Clean-up

Table II-6 defines the format of the design definition layer state table:

Table II-6: Design layer's states definition table

State's name	Action	Specifications	
[Name]		**Input**	
		Precondition	
		Output	
		Post-condition	
		Exceptions	
		Comments	

And the design layer states definition table for the Cruise Control example will be:

Table II-7: Cruise Control example design states definition table

State's name	Action	Specifications	
Constant Speed	Entry	**Input**	V_0 – set point
		Precondition	Vmin ≤ V_0 ≤ Vmax; set point in valid range for control
		Output	Call controlTask.start()
		Post-condition	controlTask running && accelerating task does not run;
		Exceptions	V_0 < Vmin – no output; V_0 > Vmax – no output.
		Comments	ControlTask keeps car speed close to set point
	Do	**None**	
	Exit	**Input**	-
		Precondition	-
		Output	Call controlTask.stop()

		Post-condition	Control task not running && accelerating task does not run.
		Exceptions	-
		Comments	Stop control task.
State's name	**Action**	**Specifications**	
Accelerating	Entry	**Input**	V_0 – set point
		Precondition	Vmin ≤ V_0 ≤ Vmax; set point in valid range for control
		Output	Call controlTask.start(); call acceleratingTask.start()
		Post-condition	controlTask running && acceleratingTask running
		Exceptions	V_0 < Vmin – no output; V_0 > Vmax – no output.
		Comments	ControlTask keeps car speed close to set point as it is being increased at constant rate by acceleratingTask.
	Do	**None**	
	Exit	**Input**	-
		Precondition	-
		Output	Call controlTask.stop(); call acceleratingTask.stop()
		Post-condition	Control task not running && accelerating task not running;
		Exceptions	-
		Comments	
State's name	**Action**	**Specifications**	
Decelerating	Entry	**None**	
	Do	**None**	
	Exit	**None**	
State's name	**Action**	**Specifications**	
No control	Entry	**None**	
	Do	**None**	
	Exit	**None**	

Note: from this table we see clearly that Decelerating and No control states are functionally identical and could be reduced or combined to one.

The third option—control variable values is documented like it is at the requirements level, see Table II-3 above and Table II-4 for the Cruise Control example.

(1) Verification

The design layer for each state is validated by analysis, showing how it meets the requirements layer. E.g. if the requirement for the Constant Speed state, from its state invariant is that the car's speed is equal to the set point, a design as specified by Table II-7 above is verified by documenting that since the control task runs in this state, it keeps the speed at the set point (up to the control module accuracy), and since the accelerating task does not run, the set point is fixed.

(2) Formal requirements to design model

A more formal model for the requirements to design transition is available for developers that use invariants to specify conditions at the context, outside the FSM, state invariant, and also use pre/post-conditions for state actions, as Figure II-3 depicts.

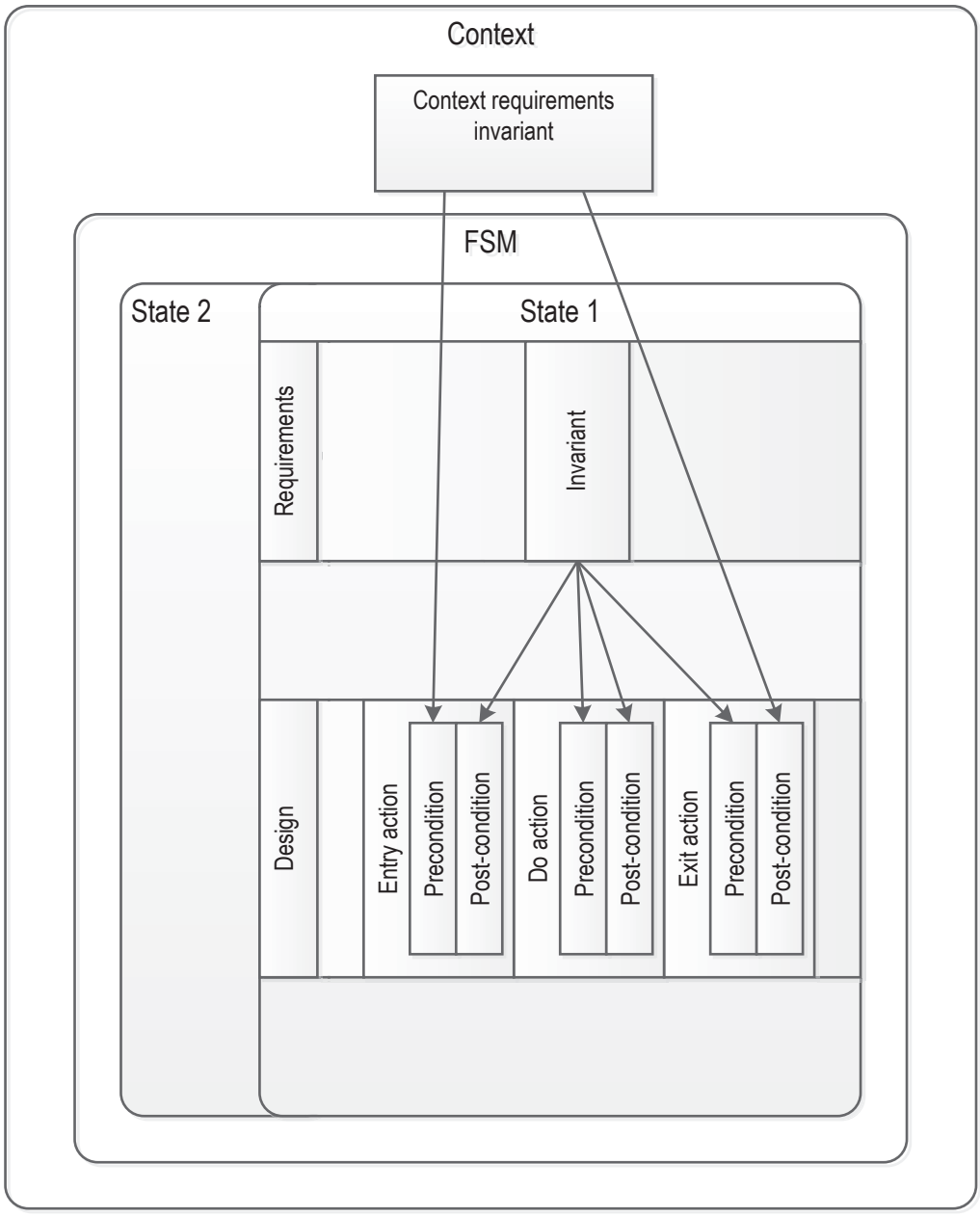


Figure II-3: Logic specifications relations between requirements and design

Requirements Level

(a) The FSM runs in some environment: a process, module, etc. That environment might have some invariant that continuous conditions in this context – the *context invariant*.

(b) When in a specific state, the **state invariant** specifies the conditions of the control structure and through it, the module controlled by the KFSM. The state invariant is stronger or more stringent than the context invariant—the specific state conditions.

Design level

(c) At the design level we specify the entry do and exit actions, by specifying their pre-and-post conditions.
(d) When we enter the state, we can only know that the context invariant applies. So the precondition to the entry action is the context invariant.
(e) The entry function's job is to bring the system initially to meet the state invariant, so that will be its post-condition.
(f) The do action runs periodically and has to maintain the state invariant; so its precondition and post-condition are the state invariant. The assumption here is that the state invariant has some "wiggle room" or tolerance, like in control systems, and the do action repeatedly checks if the control structure gets close to the boundaries and pushes it to the center. In cases where the state invariant is completely discrete, there is no creeping of the system towards violating it, there is no need for the do action.
(g) The exit action starts with the system complying with the state invariant, so it can be used as its precondition. It cleans up, and on completion has to meet only the weaker context invariant, so that will be its post-condition.

This model builds a whole set of actions specifications out of the context and state invariants.

4. The Implementation Layer

Two parts are to implementing a functionality specified by a KFSM model: implementing the KFSM itself—switching between states in response to events, and implementing its states' actions—the interface between the KFSM and the application code.

The KFSM "mechanism" itself, which is basically the transitions and the calls to the different actions can be implemented in different ways. First it can be coded by the developer; but the complexities associated with the Statechart and KFSM additional features makes this is a risky proposition. The second option is to use an FSM code generator like the FSM compiler (FSMC) [7], if one was developed for KFSM. The resulting code is likely to be more solid, depending on the code generator's quality. The third option is to develop or use an off-the shelf classes framework that implement the KFSM mechanism that can construct a KFSM object from its definition file—a *KFSM engine*. Part of object construction requires to bind the state actions and the control structure to that object. KFSM provides such a KFSM engine implementation. Its design considerations are described in section IV.

5. The Test Layer

KFSM can be instrumental in system or integration level test. Coming up with a set of test cases that has a good coverage, is a pervasive challenge to the verification team. For the part of the functionality that was developed based on the KFSM model such test case scenarios can be generated by the model as FSM productions – a set of state sequences and the event sequences that generated them. That set, produced per the KFSM model, can exhaustive to the desired scenario length, can be used to test the implementation. In cases where the control structure comprises control variables only, the FSM output for each scenario step can be verified automatically.

C. State timeout

To prevent the system from "getting stuck" in a state, waiting for an event forever, states must have a timeout defined. A "timeout" event is generated if that timeout at the state was reached. Terminal states should not have a timeout specified, and in some special cases, non-terminal states also do not require a timeout; the consideration that justify those cases should be documented as part of the general FSM model. The timeout attribute may therefore have a value of "none" and an attached justification text, documenting why that timeout value or no timeout was chosen.

D. Exceptions

Error or exception handling is a major subject in software development. Mishandled and unhandled exceptions account to about 50% of software defects by some accounts. When an exception happens its handling require to change process flow by taking step/s that are different than those of the nominal case. A model that specifies or/and executes process flows should take exceptions into account and handle them. Key FSM offers such a mechanism:

(a) Exceptions created (thrown) by a state action are added to the event list as a state-specific events; no point to consider their effect on other states where they can't be created,
(b) Exceptions in functions/methods that perform transition action or event guard conditions form additional guard conditions on that transition path. See Figure II-4.

Exception transitions looks quite complex as Figure II-4 illustrates, in practice however this part of the transitions scheme should end up being reasonably manageable; if it does get very complex, it is an indication of over complexity being built into the model, and an opportunity to correct that before coding.

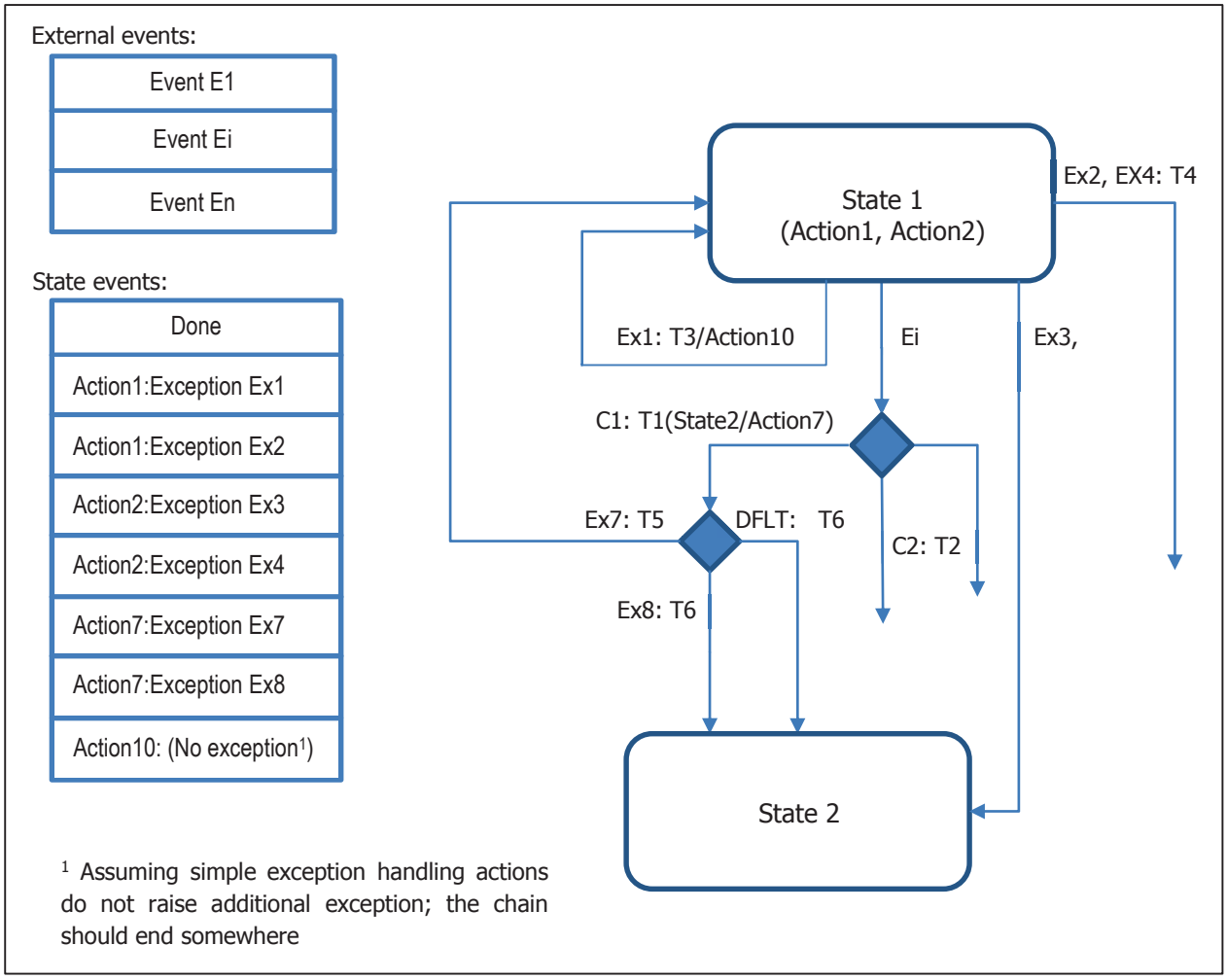


Figure II-4: Exceptions handling by KFSM

Notes: State1 actions: Action1 and Action2 raise exceptions Ex1 – Ex4. Ex1 is defined to cause transition T3 back to State1, after performing action A10 that post a message to the user/actor. Ex2 and Ex4 cause transition T4 without an action, maybe to an error state. Event Ei causes conditioned transitions. Condition C1 causes transition T1 that involves Action7 that in turn may throw exceptions Ex7 and Ex8. Those are taken as conditions that cause transitions T5 and T6 correspondingly.

E. Key FSM Transitions Model

Here we define KFSM's transition features: those that are part of the Statechart model are consider and labeled as baseline features, and those that are unique to Key FSM.

1. Initial state/s.

Normally and according to the formal FSM model (see [3]) there is only one initial state. Practically though, the initial state may be a function of certain conditions. Key FSM defines an internal "fixed" KeyFsmStart state and a "constant" start event (KeyFsmStart), with optional guard conditions like for any event transition. This provides the flexibility of starting at different states per guard conditions. See Figure II-5.

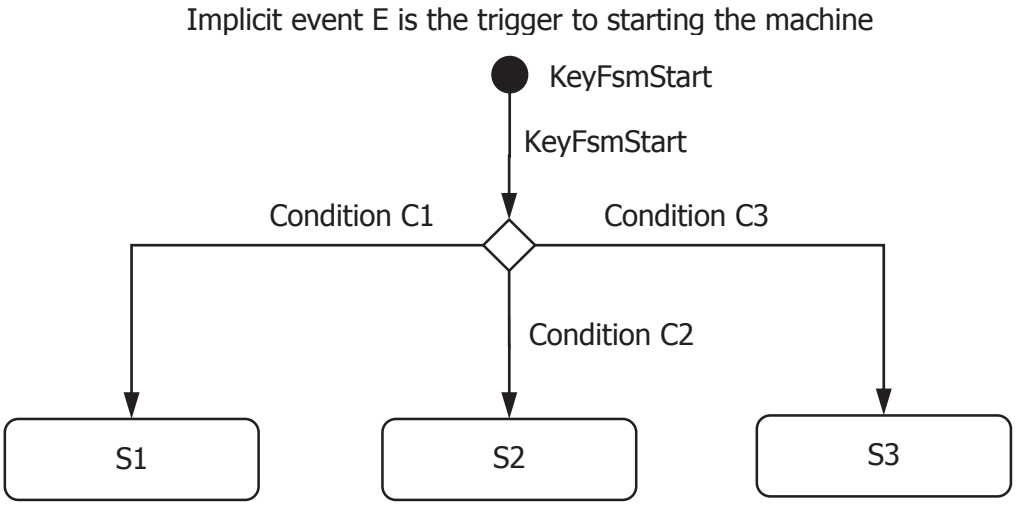
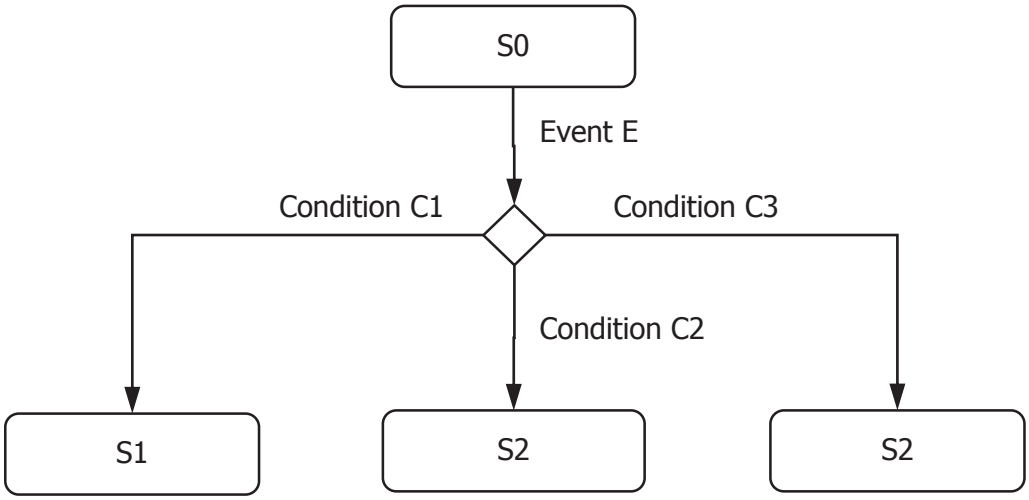


Figure II-5: Conditional start at several states.

2. Conditioned transition (baseline, Statechart)

Various transition may be defined to apply a transition when an event happens:



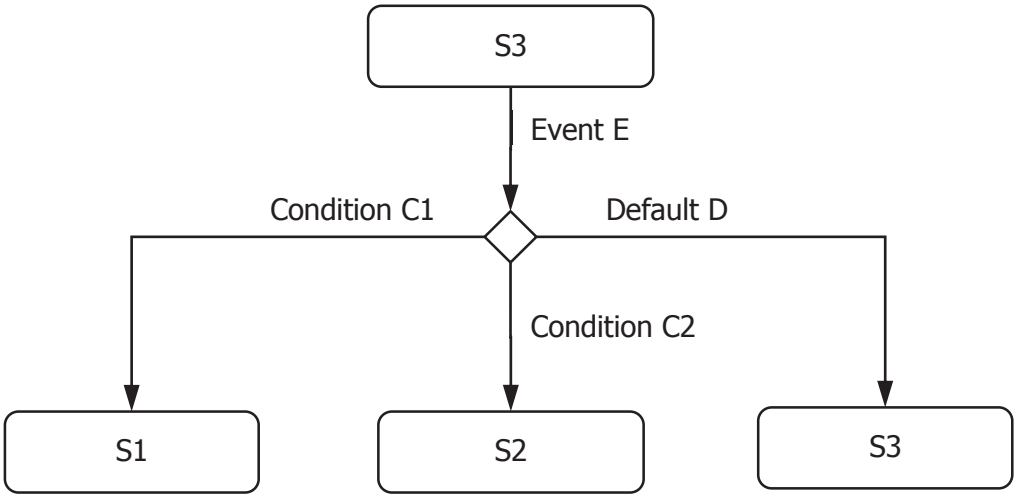
Correct conditions rules – they should cover all their variables' space, and disjoined so there is no more than one is true at a time:

(i) C1 U C2 U C3 = TRUE;

(ii) (C1 Λ C2 = FALSE) Λ (C2 Λ C3 = FALSE) Λ (C1 Λ C3 = FALSE)

3. Default condition

One of the conditions may be defined as "Default," evaluating as true for all cases that the other conditions don't.



Here the only correct conditions rule that remains is the disjoint of C1 and C2:

(i) C1 Λ C2 = FALSE;

Note: Using the default condition may be a dangerous feature or a one that adds safety. If all conditions are equal and meaningful, this is dangerous as it gives the designer an easy way to ignore some conditions and let them fall into the default category. If only a few conditions are unique cases, it is a safe way to ensure all other situations fall into the default category and are not ignored.

4. History restart (baseline, Statechart)

The start or entry point to a KFSM may be defined as "History" or (H), switching to the state the system was at when execution of the KFSM stopped. This makes most sense for a sub FSM that runs in a state as depicted in Figure II-6, as that upper state (S0 in Figure II-6) may be exited and re-entered. KFSM specifies the following with regard to such state sub-FSM structure:

(a) When the state containing the FSM is exited, the FSM is paused and it continues when the state is re-entered,

(b) Events received by the upper FSM are checked if they are defined for that FSM, and if not are directed to the sub-FSM; it is highly recommended (although not a rule) that the same events will not be defined for both,

(c) When the sub-fsm with a history entry is paused, its do-action is stopped from repeating (although it should complete if perfroming), but its exit action is not performed, since it will continue from that state on re-entry,

(d) On re-entry, a sub-fsm with history re-entry continues the do-action, skipping the entry-action.

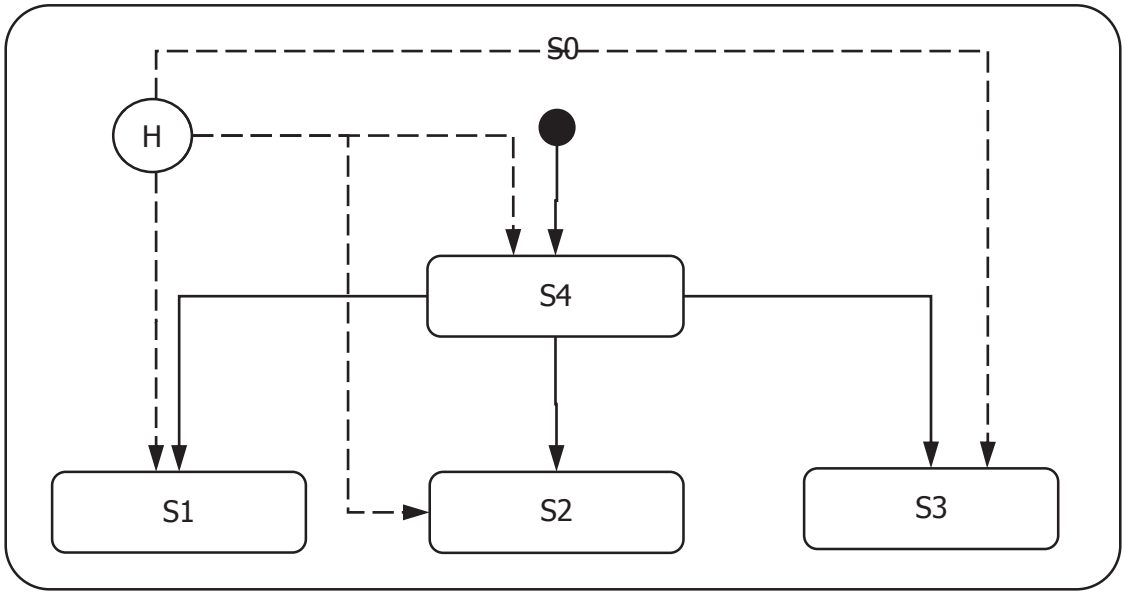


Figure II-6: History entry to a state's sub-FSM

5. Previous state

Figure II-7 illustrates a situation where transitions from several states (S1 to S3 in this case) lead to a common state (S4), and an event E1 (e.g. cancel) is to cause transitions back to the original state. A classic FSM requires to define a distinct state for the transition from each of the original states, so the FSM knows were to return to. This makes the FSM quite cumbersome in some cases; instead the KFSM's transitions scheme defines a shorthand of "previous state."

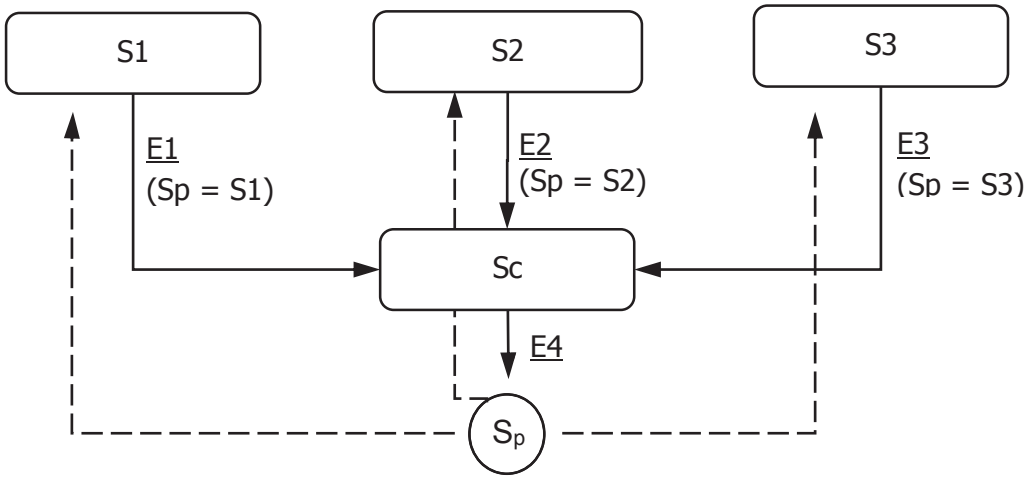


Figure II-7: Transition to previous state (S_p)

The ***previous state*** Sp is defined as the state the system was at before the current state Sc. A transition may define the next state as Sp – the previous state.

6. Destination State

In some cases an event (or different events) trigger transitions from different states to the same state, where a second event should cause a transition to different new states depending on the previous state. The mechanism KFSM offers to support these cases better while simplifying their specifications is the ***destination state***. States may define a destination state through their definition XML file, no need to write code to do it; that destination state is used next if a transition is defined to "destination state." To prevent errors, this destination state is kept only for the next state and erased later, so very old definition can't be used. See Figure II-8.

Note: this is equivalent to having a separate state for each original state, leading to a distinct destination state; the destination state feature reduces clutter and duplication in the KFSM definition.

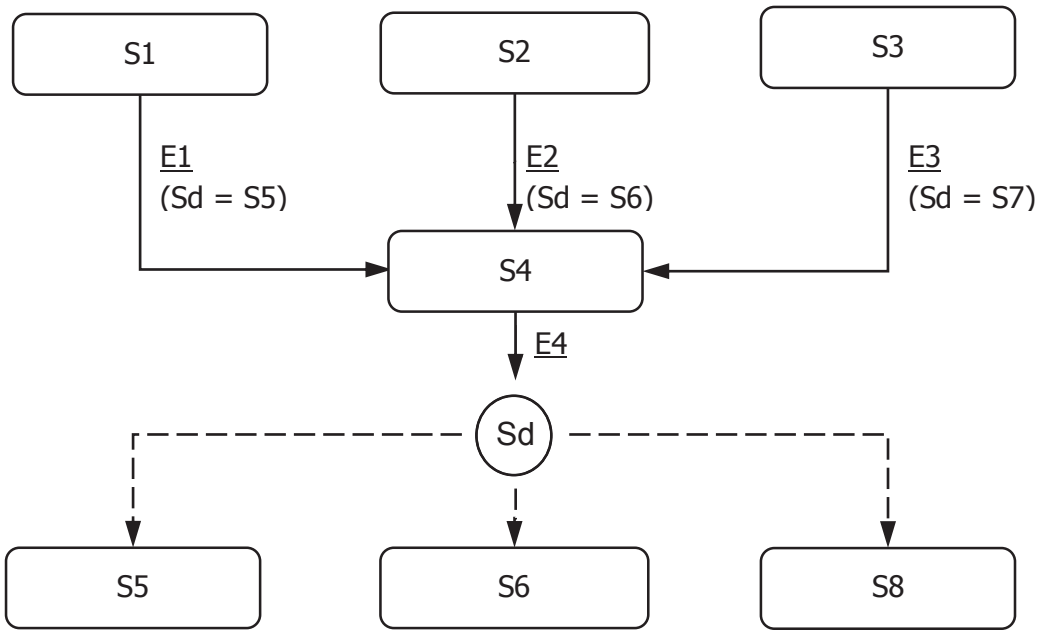


Figure II-8: Destination state

Since at least some of the events that activate the FSM are asynchronous to its own operation nation state representation

F. Key FSM Evaluation

A major advantage of using a meta-model is that it can be checked for correctness using model-type specific rules. The FSM, as a mathematical and formal model supports such analysis and Key FSM specifically offers ample rules for evaluating a specific KFSM model. We divide those correctness rules to two types: structural and content rules.

1. Structural KFSM Correctness Rules

(1) Initial state/s
At least one transition (no guard condition) exist from the KeyFsmStart state to an initial state. If more than one transition exists from the KeyFsmStart, they all have to be guard-conditioned.

(2) All states are defined
A state exists with the name specified as next state of any transition.

(3) All states are reachable
Every real state (excluding the ArenaFsmStart state) has a transition to it.

(4) Complete transitions definition
Every event defines for each state either:

a. A transition – next state and an optional transition action, or
b. Void transition
c. Reasoning documentation for both of those cases.

(5) Timeout (I)
A timeout attribute with time value or "none" symbol value is specified for every non-terminal state, accompanied by a rationale documentation field.

(6) Timeout (II)
For each state with a timeout definition, a transition out triggered by a "timeout" event must exist.

(7) Transition conditions (I)
If an event appears as the trigger of more than one transition from the same state, all those transitions must have a guard transition defined, conditioning the triggering event.

(8) Transition conditions (II)
All conditions Ci on same events of a state must satisfy:

No default condition:
(a) they are all disjoined: Ci AND Cj = FALSE, when i ≠ j (otherwise, the transition is ambiguous),
(b) Their union is TRUE: C1 OR C2 OR ... Cn = TRUE (there is no harm in conditioning an event so that under some conditions it does not trigger a transition).

Default condition specified:
(a) Only one conditioned transition has a default condition specified,
(b) All other (not default) conditions are disjoined: Ci AND Cj = FALSE (otherwise, the transition is ambiguous).

(9) Exception events
Every exception defined for state's actions (entry, state and exit) defines a state specific event.

(10) Exception conditions
Every exception defined for a transition action is a condition on that transition's event.

(11) Terminal state action
No state action should be defined for a terminal state – since this will cause the action to be performed forever – a warning should be issued for such a condition, to examine it.

(12) Terminal states (I)
Any state that has only exception triggered transitions from it is defined as terminal and vice versa.

(13) Terminal State (II)
An exception triggered transition from a terminal state can only lead to that same state. (Its benefit is triggering a transition action for logging error messages, and invoking the state's entry action again. Note that an exception at a terminal state is possible only if it has an entry or a do action defined.)

(14) Terminal states (III)
No timeout nor exit action are defined for a terminal state.

(15) Terminal states (IV)
Only exceptions event (id ends with "Exception") is allowed for a terminal state.

2. Dynamic correctness rules

The following rules depend on run-time data; those that can't be evaluated using specification expressions should be evaluated in runtime:

(16) Transition conditions III (substitute rule (8) above if can't be implemented as a static check)
(1) All non-default conditions are mutually exclusive: only one of them evaluates to TRUE. Otherwise – raise an ApplicationDefinitionException,
(2) One condition (including the default if defined) must evaluate as TRUE.

III. The Event-State Analysis (ESA)

Often, FSM models are constructed intuitively and then reviewed by a team of experts; rarely such a model does not miss some transitions or even states. ESA is a semi-formal process that guides the analyst to find and complete those missing parts, improving its completeness and correctness considerably. In addition, it also leads the analyst to document decisions made in constructing and completing the model and their rationale, improving system maintainability.

Formally or mathematically, it is based on the FSM definition's transition mapping function: S x E → T. This function defines a transition for every combination of states and events, and this is the key to the method.

The input to the ESA process is an FSM model that has been discussed and reviewed—the ***nominal FSM***. The process modifies the FSM model—adding missing transitions and states. The initial FSM being analyzed can be empty—the initial state alone (required), the process "completes" this empty model, in essence **constructing** the FSM.

A more formal definition of the event-state analysis:

Input: FSM model: A set S of states, a set E of events, and a set T of transitions, where Ei and Sj maps to Tk. S can be an empty set—we start with a "blank slate" and build the FSM model using this process.

Output: The same FSM model, updated and completed: S', E', T'.

Step 1: Exhaustive events list

Verify that the events list of the FSM spec is complete. This is done by reviewing the list with SMEs and colleagues. In some cases the overall system has a master events list, at least of external events from the event-response diagram which is the highest level system description, those events should be reviewed in order to see if any affect the analyzed FSM's control structure and are therefore relevant.

Add the following events:

- For each state that has a timeout defined, add a "timeout" event,
- For each state that has state actions, add exception events, event key: exception-key+exceptionClass,

Step 2: Applying events

Iterate through the FSM's states list, for each state Si:

Iterate through the events list, for each event Ej:

Consider if the event should trigger a transition from the state:

a. Always or under certain conditions? For each transition with a guard condition or a transition action – add additional conditions for each

exception that can be thrown by the guard condition function and by the action method.

b. To which state? An existing state or a new one?

With those decisions, revise the appropriate transition table line or add one or more lines to the state transitions table:

	State	Event : Guard cond.	Next state	Documentation, Comments
. . .				
0.	Si	Ej : C1		Reason for the guard and transition
1.	Si	…		- " -
2.	Si	Ej: Ck	Sj	- " -

Some comments to this step:

(a) The input model's set of states is never empty, it can however include only one state – the "FSM-start pseudo state. In this case we only apply the "start" pseudo event to it, consider if any guard conditions are required, and so define the initial transition/s to the initial state/s. In this case we use the ESA to construct the FSM rather than analyze and improve it.

(b) In some cases we realize that an event should transition the FSM from the examined state Si, but none of the other states is appropriate to transition to. We need a new state for the system to be in. We define that state and add it at the end of the states' list and table:

	State name	Description	Specification(1)	Comments
3.	New state	Does something new	Var1 = C1; Var2 = C2; … Terminal: N Timeout: N (or the time)	Document any additional information and consideration.

(1) The chosen specification notation (text specifications, state variables or state invariant for requirements, action specifications for design) is used for all states. The control variable constraints method is used here.

(2) Since the new state is added at the end of the states' table, the state iterator will reach it and will iterate all the events while it is the examined state.

(3) In case the RA can't answer the question what transition should be triggered by the event at the given state, a subject matter expert should be consulted. If there are too many cases like that, or iv the RA does not feel confident to take those decisions in the first place, the whole process can take place in an elicitation meeting with the SME's.

End of procedure, the FSM model has been updated with new transitions and states.

So what is this "magic" method? The essence of it is: check EVERY event against EVERY state, consider if it requires a transition and to which state. This semi-mechanical process finds many missing transitions and states in the starting nominal FSM.

The key to succeeding with ESA is having a list of ALL the events that affect the FSM's object. Every effort must be made not to miss any such event. If in a later point an event that should be considered but was missed is found, it can be added as will be shown later.

IV. KFSM Design and Implementation

A. Requirements

Key FSM implementation and its design and verification is done to a list of requirements. This paragraph defines the requirements for the Key FSM reference implementation.

1. Transition model

Key FSM implements the transition features described by section II.E above and by the UML FSM specifications.

2. State content

Key FSM implements state content as specified by section II.B above.

3. Control Structure

Key FSM implements the control structure concept as specified by section II.A above.

4. Scalability and Efficiency

Key FSM reference implementation is required to have linear scalability, and reasonably minimize instance memory and processing requirement (invest reasonable design effort).

5. Integration

Following are considerations relating to the manner in which Key FSM supports its integration into a system.

(1) KFSM as a Member

A class representing a processing unit of any kind can have a Key FSM object or instance as a member, signal events to it causing it to transition between states, apply its defines state action, control the control structure, or use State property like its id or name to control its processing.

(2) KFSM State as a Member

A class may have a member variable of class KeyFsmState – which is the state of a KFSM instance. The class may apply events to the KeyFsmState variable causing its KFSM to transition, which will replace the content and active parts of the KeyFsmState variable: properties (like name, description), control structure values, and do action task and timeout timer. Its new entry task was already invoked by the transition.

6. API's

Both the KeyFsm and KeyFsmState classes implement the KeyFsmApi that includes the following methods:

(1) Boolean event(KeyEvent event) throws KeyException; where:

- The method returns true iff the event caused a transition,
- KeyEvent contains:
  (a) a key for identifying the type of the event,
  (b) a unique id, identifying the event instance,
  (c) data required to process the event by state actions.
  (d) Actor and session that created the event and whose access rights are to be used to access the control structure or other resources required by state and transition actions.
- KeyException is defined by the integrating framework to satisfy its exception handling framework.

(2) String getProperty(KeyFsmPropertyKey propertyKey, KeyContext context) throws KeyException; where:

- KeyFsmPropertyKey is an enumeration type identifying properties available by the KFSM,
- KeyContext provides access to system services like logging and to session and actor information including access rights to the various properties,
- KeyException is defined by the integrating framework to satisfy its exception handling framework.

B. Efficiency and scalability design

This section describes the design of the KFSM implementation. The main consideration that guides the design is efficiency and scalability.[1] Here are some considerations related to efficiency:

(1) The KFSM as described holds many "members" or fields; many relate to requirements and design, and are not always useful for runtime.
(2) The current state's reference/s to the control structure do action task and timeout timer are the only instance specific members; all the rest are common to all instances.

[1] Scalability is the type of load growth with the number of instances. Linear growth is a "neutral" scalability, polynomial and faster growth is poor scalability, and logarithmic and n'th root are good scalability. Efficiency is the constant that multiplies such a normalized curve. The lower that constant is the higher and better the implementation efficiency, meaning that less server resources are needed for the same number of simultaneous user or instances.

From these considerations we derive the following efficiency design guidelines, its scalability curve will be linear with the number of instances, see Figure IV-1.

(3) A "ReferenceKeyFsm" class constructs a referenceKeyFsm singleton object for each type of KFSM, i.e. KFSM of a specific definition (XML file).
(4) The ReferneceKfsm singleton contains state singletons for all its states of class ReferenceKeyFsmState, that hold transition singleton. This is a KeyFsm instance factory—when a KeyFsm instance is required, the RefernceKeyFsm constructs one, which is not its sub-class.
(5) A KFSM instance contains only an instance of the current state that is cloned from a state singleton, and a reference to the control structure of the object controlled by the KFSM.
(6) The KeyFsm class has an event() method for signaling an event to the KFSM, a currentState() method for returning the current state and a property(key) method for returning a KeyFsm property like its name, description, etc.
(7) A KeyFsmState instance is constructed by its factory ReferenceKeyFsm when a new current state is needed—during a transition.
(8) The KeyFsmState class public interface also contains an event() methos; this is done so a KeyFsm's current state can be a member of an object that signals events to it, the KeyFsmState object passes uses the KeyFsm instance's event() method. In addition, the KeyFsmState class also supports the keyFsmProperty(key) method so KFSM property can be obtained.
(9) When a record field or entity's property is an FSM state, its type is specified as KeyFsmState(fsmKey), defining it unambiguously.

This design, as depicted by Figure IV-1, incurs very little overhead per instance; only the small KFSM and KeyFsmState objects are constructed and maintained for each Key FSM instance.

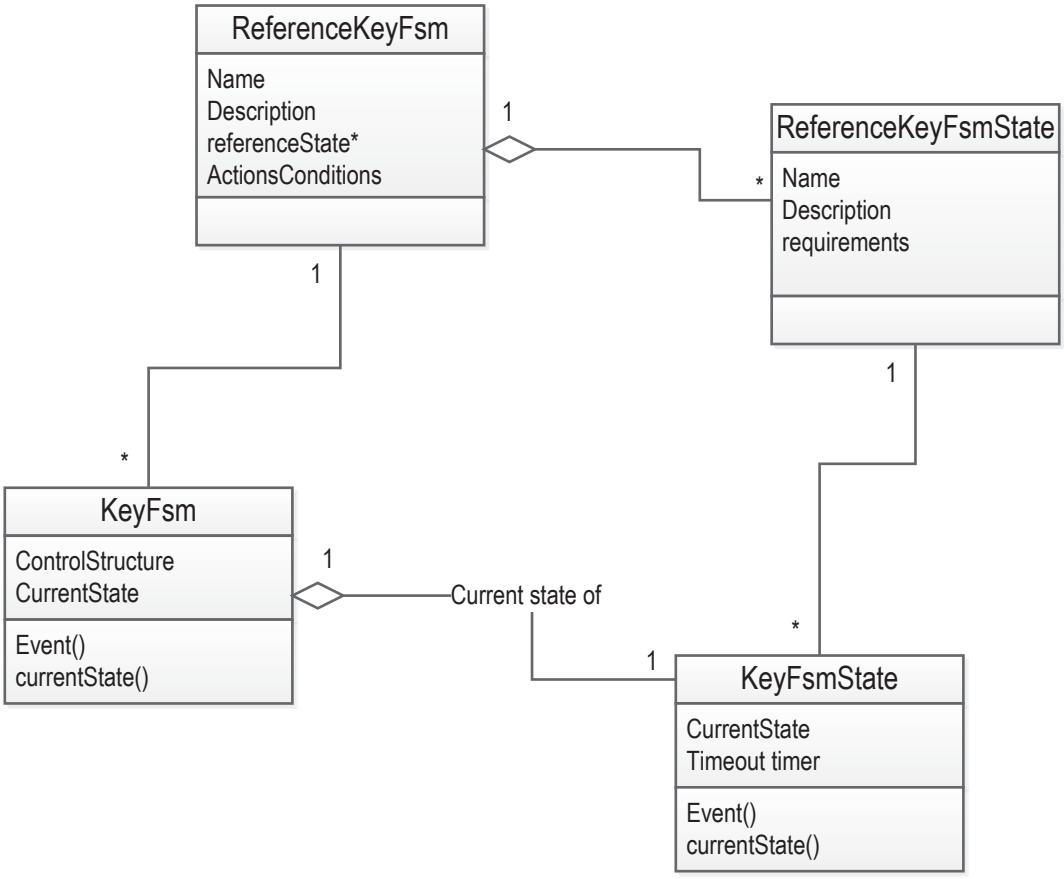


Figure IV-1: Class diagram representation of the Key FSM high level design

C. Implicit control-variables assignment

For control structure that includes only control variables, and their values are specified for each state, either at requirements or design levels, the developer may select this option and have the KeyFsmState, using its ref to the KeyFsm instance, set the values of control variables for the current state, creating a complete implementation without the need for any application programming.

V. Further Work - Key FSM Implementation

A reference implementation of the design outlined above is work in progress, including a utility for editing and checking a Key FSM.

References

[1] A. Avnur *Finite State machine for real-time software engineering*, IEE Computing & Control Engineering Journal, Nov. 1990, pp 273.

[2] Cruise control FSM requirements modeling, http://www.aa-sw-dev.com/KeyFSM/Cruise%20control%20requirements%20FSM.htm

[3] Stanford University, Basics of Automata Theory, http://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html,

[4] D. Harel, Statecharts: A Visual Formalism for Complex Systems, Science of Computer Programming, 8 (1987), 234-274.

[5] IREB Glossary, http://www.ireb.org/fileadmin/IREB/Download/Homepage%20Downloads/IREB_CPRE_Glossary_16.pdf

[6] Mavin, A. at al, Easy Approach to Requirements Syntax (EARS), Requirements Engineering Conference, 2009. RE '09. 17th IEEE International.

[7] SMC – The States Machine Compiler, http://smc.sourceforge.net

[8] University of Toronto, Dept. of Computer Science, *Requirements Modeling Lecture*, slide 5, http://www.cs.toronto.edu/~sme/CSC340F/slides/09-modelling.pdf

[9] Karl E. Wiegers, Software Requirements, Microsoft Press, 2nd Edition.

[10] Wikipedia, Finite state-machine, http://en.wikipedia.org/wiki/Finite-state_machine

[11] Wikipedia, UML State machine, http://en.wikipedia.org/wiki/UML_state_machine

Appendix 1: Cruise Control Key FSM Example

This simple system example to illustrate various aspects of creating and using a requirements FSM and then following up with a design.

A. Requirements FSM, Baseline or Nominal Version

After analyzing the system, the following baseline KFSM was defined:

Key FSM Definition

1. Events

Event	Description
E1	Start – start constant speed control,
E2	Stop – stop speed control
E3	Up pressed – increase speed
E4	Up released – got to desired speed
E5	Down pressed – want to reduce speed
E6	Down released – got to desired speed
E7	Throttle error

2. States: General and Requirements

State (name / description, attr.)	Specifications	State variable constraints	State invariant
No control	The system does not control the car speed.	controlOn: False; Vc: ≥ 0;	Vc: ≥ 0;
Constant speed	The system shall control car speed as reported by the input variable Vc, keeping it constant by controlling throttle position.	controlOn: True Vc: V_0 ± DV.	V_0 – DV < Vc < V_0 + DV
Accelerating	The system shall continue to control the car speed, the set point speed is incremented by DV initially, and every Dt sec. afterwards. DV and Dt are constants.	controlOn: True; Vc: V_0 + DV (1 + t/Dt); DV and Dt are constants.	$V_1 = V_0 + DV(1 + t/Dt)$ AND V_1 – DV < Vc < V_1 + DV
Decelerating	The system shall stop controlling the throttle, letting it drop to idle and allow the car to coast to lower speed.	controlOn: False; Vc(t + dt) < Vc(t);	Vc(t + dt) < Vc(t);

3. Transitions

	State	Event : Guard cond.	Next state	Documentation, Comments
1.	No control	Start	Constant speed	Obvious, nothing to add.
2.	Constant speed	Stop	No control	Obvious, nothing to add.
3.	Constant speed	Up pressed	Accelerating	Obvious, nothing to add.
4.	Constant speed	Down released	Decelerating	Obvious, nothing to add.
5.	Constant speed	Throttle error	No control	A SME decision to stop control on error.
6.	Accelerating	Up released	Constant speed	Obvious, nothing to add.
7.	Accelerating	Throttle error	No control	A SME decision to stop control on error.
8.	Accelerating	Timeout	No control	SME decision not to let the driver accelerate indefinitely.
9.	Decelerating	Down released	Constant speed	Obvious, nothing to add.
10.	Decelerating	Throttle error	No control	A SME decision to stop control on error.

And the corresponding state-transitions diagram:

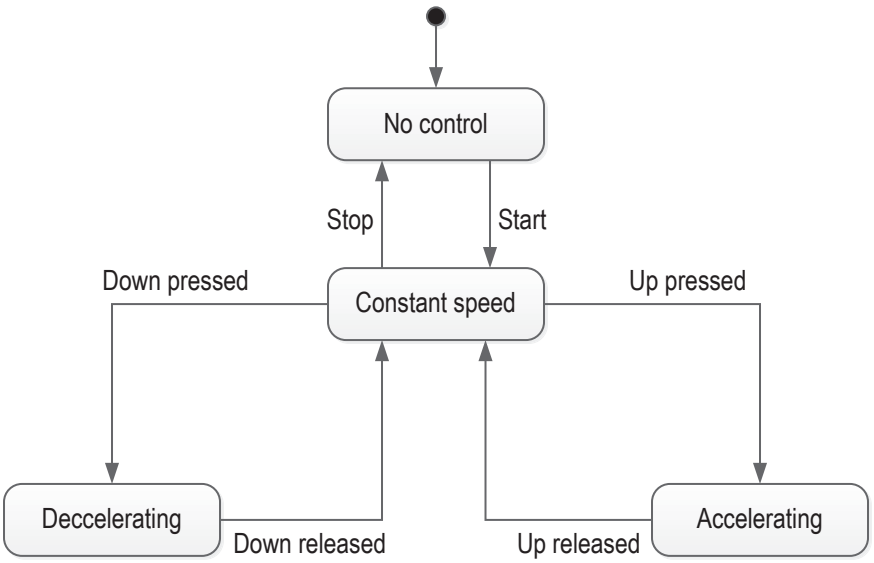


Figure 2: Nominal requirements FSM

B. Event-state analysis

The Event-state analysis (ESA) guides the analyst or engineer to complete the FSM model, identifying missing states and transitions. The baseline KFSM version is used as the input to the ESA procedure (described in section III above). The result is an updated KFSM definition:

Key FSM Definition

1. Events

Event	Description
E1	Start – start constant speed control,
E2	Stop – stop speed control
E3	Up pressed – increase speed
E4	Up released – got to desired speed
E5	Down pressed – want to reduce speed
E6	Down released – got to desired speed
E7	Throttle error
E8	Timeout
E9	Breaks applied

2. States: General and Requirements

State (name / description, attr.)	Specifications	State variable constraints	State invariant
No control	The system does not control the car speed.	controlOn: False; Vc: ≥ 0;	Vc: ≥ 0;
Constant speed	The system shall control car speed as reported by the input variable Vc, keeping it constant by controlling throttle position.	controlOn: True Vc: $V_0 \pm DV$.	$V_0 - DV < Vc < V_0 + DV$
Accelerating	The system shall continue to control the car speed, the set point speed is incremented by DV initially, and every Dt sec. afterwards. DV and Dt are constants.	controlOn: True; Vc: $V_0 + DV\ (1 + t/Dt)$; DV and Dt are constants.	$V_1 = V_0 + DV(1 + t/Dt)$ AND $V_1 - DV < Vc < V_1 + DV$
Decelerating	The system shall stop controlling the throttle, letting it drop to idle and allow the car to coast to lower speed.	controlOn: False; $Vc(t + dt) < Vc(t)$;	$Vc(t + dt) < Vc(t)$;

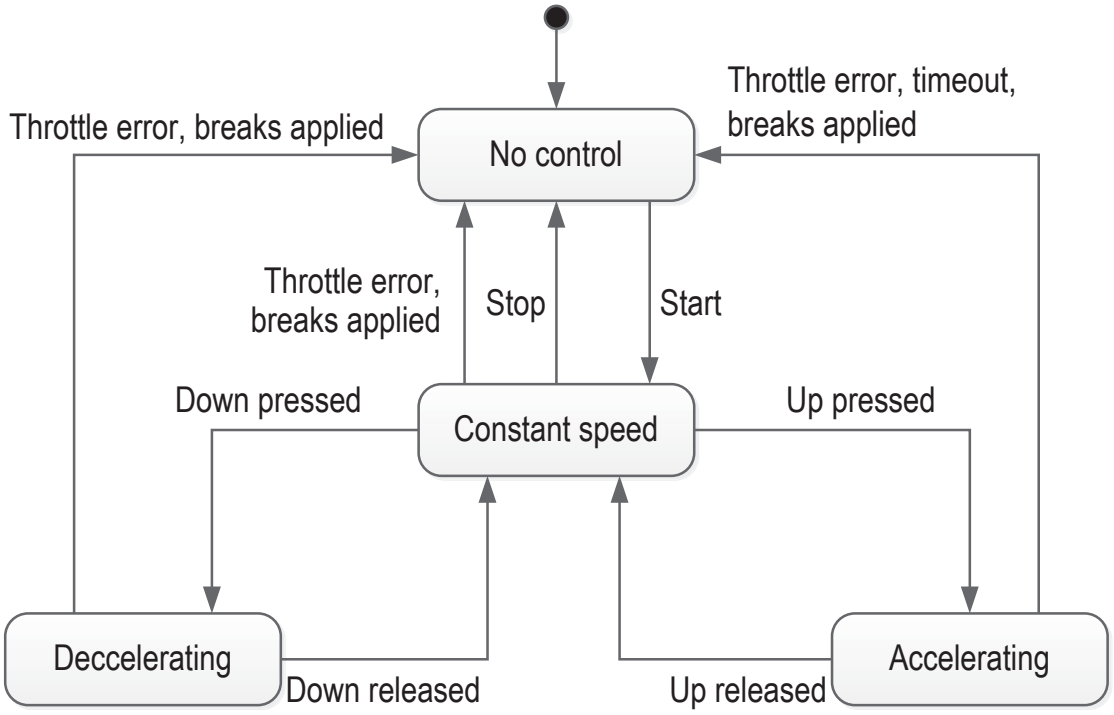
3. Transitions

	State	Event : Guard cond.	Next state (a)	Documentation, Comments
1.	No control	Start	Constant speed	As intended.
2.	No control	Stop	No transition	Irrelevant at this state.
3.	No control	Up pressed	No transition	Interaction design decision not to respond to up and down button at when not controlling speed.

4.	No control	Up released	No transition	- " -
5.	No control	Down pressed	No transition	- " -
6.	No control	Down released	No transition	- " -
7.	No control	Throttle error	No transition	Irrelevant at this state.
8.	No control	Breaks applied	No transition	Irrelevant at this state.
9.	No control	Timeout	No transition	No timeout.
10.	Constant speed	Start	No transition	Irrelevant at this state.
11.	Constant speed	Stop	No control	As intended.
12.	Constant speed	Up pressed	Accelerating	As intended.
13.	Constant speed	Up released	No transition	Impossible (pressing the button transitions to another state first).
14.	Constant speed	Down pressed	Decelerating	As intended.
15.	Constant speed	Down released		Impossible (pressing the button transitions to another state first).
16.	Constant speed	Throttle error	No control	A SME decision to switch off control when getting an error.
17.	Constant speed	Breaks applied	No control	A SME decision to switch off control when breaks are applied.
18.	Constant speed	Timeout	No transition	No timeout.
19.	Accelerating	Start	No transition	Disabled; interaction design decision not to respond to this button at this state.
20.	Accelerating	Stop	No transition	Disabled; interaction design decision not to respond to this button at this state.
21.	Accelerating	Up pressed	No transition	Not possible; already pressed.
22.	Accelerating	Up released	Constant speed	As intended.
23.	Accelerating	Down pressed	No transition	Disabled; interaction design decision not to respond to this button at this state.
24.	Accelerating	Down released	No transition	Disabled; interaction design decision not to respond to this button at this state.
25.	Accelerating	Throttle error	No control	SME decision to stop control on error event.
26.	Accelerating	Breaks applied	No control	SME decision to stop control when beaks applied.
27.	Accelerating	Timeout	Constant speed	SME decision to switch to constant speed.
28.	Accelerating	Timeout	No control	SME decision not to let the driver accelerate indefinitely.
29.	Decelerating	Start	No transition	Disabled; interaction design decision not to respond to this button at this state.
30.	Decelerating	Stop	No transition	Disabled; interaction design decision not to respond to this button at this state.
31.	Decelerating	Up pressed	No transition	Disabled; interaction design decision not to respond to this button at this state.

32.	Decelerating	Up released	No transition	Disabled; interaction design decision not to respond to this button at this state.
33.	Decelerating	Down pressed	No transition	Not possible; already pressed.
34.	Decelerating	Down released	Constant speed	As intended.
35.	Decelerating	Throttle error	No control	SME decision to stop control on error event.
36.	Decelerating	Breaks applied	No control	SME decision to stop control when beaks applied.

And the corresponding state transitions diagram:



The main additions here are timeout, and the additional event of breaks applied.

C. Discussion of the example

(a) The complete state transitions table has 36 rows compared with the 6 of the first, "nominal" version.

(b) Four new meaningful transitions (red rows) were found. Note that the main addition here is the introduction of timeout and breaks applied events. An exception analysis reveals that in this case virtually all exception conditions end up generating a throttle error – an event already on the list, so no need to add events for those conditions.

(c) Many events require no transition for simple reasons: either the event is impossible in this state, or a decision was made not to respond to the event and even disable it in some way. Many common FSM notations and models just leave those rows out as not-interesting; ESA requires to have those rows and specify "No transition", documenting the reasoning. This prevents accidental response when code changes are made for continued development. Reasoning documentation supports safer maintenance. Key FSM implementation uses those rows to make sure there are no transitions in those cases, preventing arbitrary responses.

(d) Some event–state combinations require expert decision what to do: which state to transition to or no transition. The analyst may present the question to a SME directly; the decision will be reviewed by the whole team later on. If there are too many such cases, an elicitation follow-up meeting to provide those decisions is justified.

(e) Some transition decisions are the result of interaction design. This interface design process still produces requirement decisions (even if it may be considered being further downstream).

(f) All rows contain rationale documentation in the documentation / comments column. Important for maintenance, when another developer considers to modify the FSM can read the reason for its definition and avoid errors. Some trivial rows are documented with "as intended" to record that this transition implements a basic system response to the event. For tracing transition decision, references to specific requirements, SME statement, etc.

D. Step 5: Design

The first design decision that has to be made is if to use an FSM to control the cruise control system. A relatively simple examination indicates that the four states of the requirements FSM are meaningful and a straight forward choice for implementation, so we decide to implement such an FSM. A design FSM with the same states and transitions serves as the specifications and model for implementation.

Next design decision is how to implement the two main system functions: speed control and acceleration. Control experts develop a control module that keeps the car's speed as close as possible to the set point that has to run every Tc mili-seconds. An acceleration module is developed that changes the set-point at the required constant rate that has to run every Ta mili-seconds. Those modules can be called periodically by the do action of the constant speed and accelerating states. A second choice is that they are called by independent tasks that poll two control variables: controlOn and accelerating and call those modules when the variables have the value 'true'. Yet a third choice is to control those two tasks by calling their start and stop methods by the entry and exit actions of the constant speed and accelerating states.

To illustrate the Key FSM capability to implement the control function of such a system completely, without a need for coding, not even of state actions, we choose the second option—

the two tasks will poll the control variables updated by the Key FSM, as depicted by the following diagram:

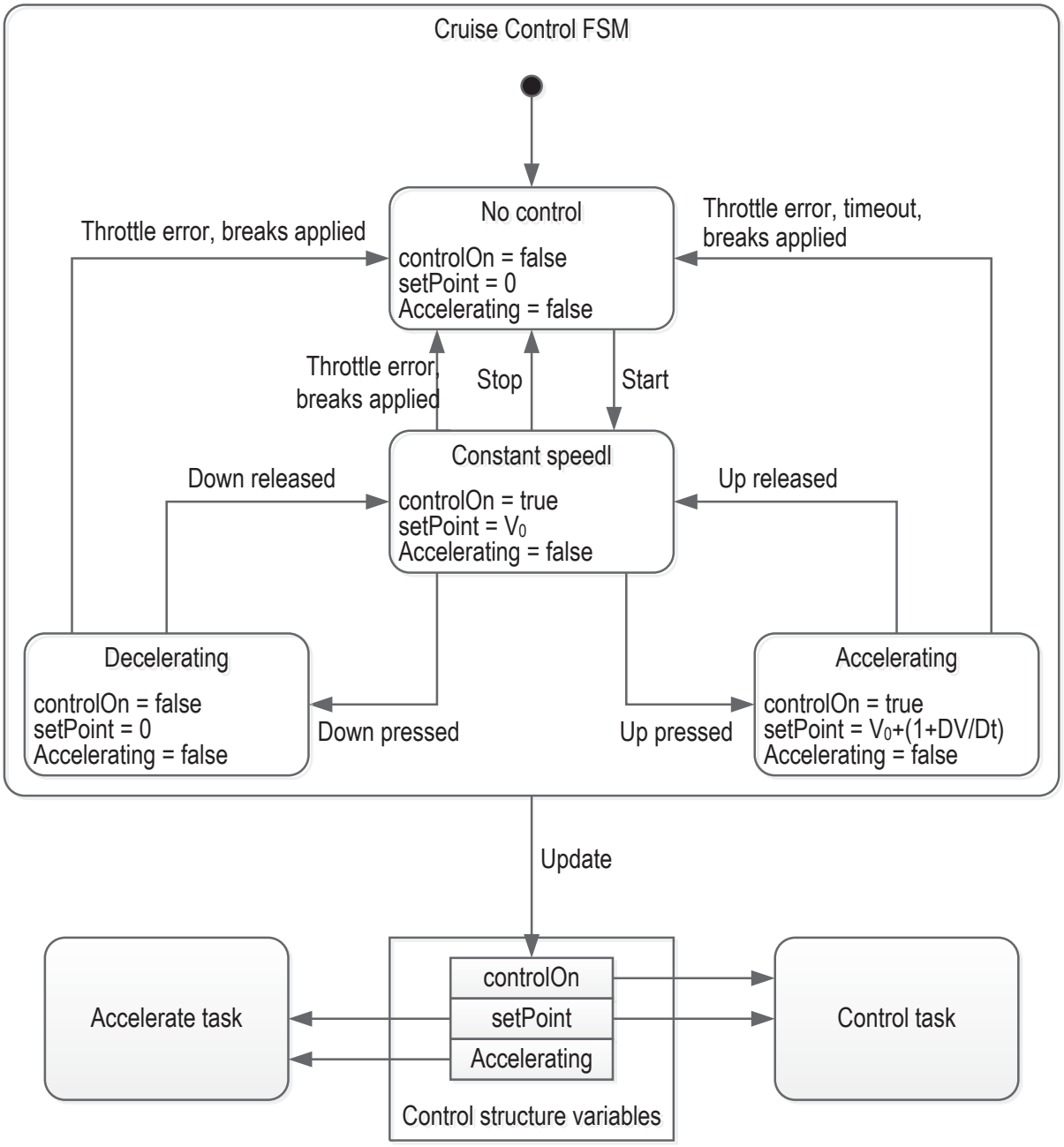


Figure 3: Cruise control design and design FSM