SAE has provided this Draft document for the SAE Committee. This document is SAE-copyrighted, intellectual property. It may not be shared, downloaded, duplicated, or transmitted in any matter outside of the SAE Committee without SAE's approval. Please contact your staff representative for additional information.

SAE INTERNATIONAL

AEROSPACE STANDARD

AS5506™ REV. D

Draft 2019-05

Superceding AS5506C

Architecture Analysis and Design Language (AADL)

RATIONALE

This Architecture Analysis & Design Language (AADL) standard document was prepared by the SAE AS-2C Architecture Description Language Subcommittee, Embedded Computing Systems Committee, Aerospace Avionics Systems Division.

The language was originally published as SAE AS5506 in 2004. The language has been refined and extended based on industrial experience as AADL V2 and published as AS5506A in 2009. The improvements focused on better support for architecture templates and modeling of layered and partitioned architectures. AADL V2.1 and V2.2, revisions that address a number of errata and minor improvements agreed upon by the committee, were published as AS5506B in 2012 and AS5506C in 2017.

This document AS5506D documents AADL V3, a major revision AADL based on industrial experience, using AADL V2.2 as baseline. This revision introduces new concepts in addition to addressing errata.

DRAFT © SAE International

SAE Technical Standards Board Rules provide that: "This report is published by SAE to advance the state of technical and engineering sciences. The use of this report is entirely voluntary, and its applicability and suitability for any particular use, including any patent infringement arising therefrom, is the sole responsibility of the user."

SAE reviews each technical report at least every five years at which time it may be revised, reaffirmed, stabilized, or cancelled. SAE invites your written comments and suggestions.

Copyright © 2016 SAE International

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of SAE.

TO PLACE A DOCUMENT ORDER: Tel: 877-606-7323 (inside USA and Canada)
Tel: +1 724-776-4970 (outside USA)
Fax: 724-776-0790
Email: CustomerService@sae.org
SAE WEB ADDRESS: http://www.sae.org

SAE values your input. To provide feedback on this Technical Report, please visit
http://www.sae.org/technical/standards/PRODCODE

TABLE OF CONTENTS

No table of figures entries found.

DRAFT
© SAE International

1. AADL DYNAMC SEMANTICS

Description

(1) An AADL specification is governed by a set of static semantics rules that specify how model elements are organized in terms of containment hierarchy, allowed relationships (connections, bindings) with other components, XXX, allowed subclauses. Dynamic semantics provide an additional set of rules to interpret the behavior of atomic AADL components and how these behaviors can be composed to describe the execution of an AADL model.

(2) The purpose of this part is document the dynamic semantics of component categories. Properties that can apply to component categories, it builds on concept introduced on the static semantics part,.

(3) An AADL model is built around an assembly of components that defines a containment hierarchy. Typical rules concern processes that combine threads that execute subprograms, schedulers configured at the level of a virtual processor that provide execution resources to threads, etc.

Processing Requirements and Permissions

(4) A method of implementation may adapt this dynamic semantics to capture a specific semantics.

Examples include specific component lifecycle, port interactions that are captured through non-standard properties.

2. SOFTWARE COMPONENTS

(5) This section defines the dynamic semantics of following categories of software components: data, subprogram, subprogram group, thread, thread group, and process.

(6) Generally speaking, software components categories have an abstract dynamic semantics: a generic automata is provided and can be later adjusted based on some properties attached to a component.

2.1 Subprograms

(7) A subprogram component represents sequentially executed source text that is called with parameters.

Dynamic Semantics

(8) Subprogram implementations and thread implementations can contain subprograms as subcomponents. The corresponding subcomponents, connected to the enclosing components features or other sibling components form a call graph, or a graph of subprogram calls, with parameters flowing through each node of the call graph using actual AADL connection semantics for features. A method of implementation can decide to turn this flow graph into a sequence of subprogram calls, or treat them as a parallel execution of subprograms.

(9) In case of a requires subprogram access the call is either local to a subprogram instance in the containing process, or remote to a subprogram instance in another process. Subprogram access connection declarations identify the subprogram instance to be called.

(10) The standard permits modeling of subprograms and subprogram calls without requiring the declaration of subprogram instances. In this case, subprogram calls may refer to subprogram classifiers and the source language processing system will determine the subprogram instance to be called. In the case of remote subprogram calls the target subprogram is identified by subprogram call properties. An `Allowed_Subprogram_Call` property, if present, identifies the remote subprogram(s) that are allowed to be used in a call binding. An `Actual_Subprogram_Call` property records the actual binding to a subprogram or provides subprogram access feature. Constraints on the buses and processors over which such calls can be routed can be specified with the `Allowed_Subprogram_Call_Bindings` property.

(11) The following control flow semantics apply to subprogram calls, when the call refers to:

- Subprogram classifier: execution by the calling thread
- Provides subprogram access of data type: execution by the calling thread

DRAFT © SAE International

- Subprogram subcomponent in calling thread: execution by the calling thread
- Provides subprogram access feature of a data component: execution by calling thread
- Subprogram access to subprogram component in enclosing thread group, process, or system: execution by calling thread
- Subprogram access to subprogram component in another thread group, process, or system: execution by calling thread
- Provides subprogram access of another thread: execution by called thread
- Provides subprogram access feature of a device: execution inside device
- Subprogram access of a processor: execution inside the processor (operating system)
- Subprogram classifier and the call has a subprogram call binding property that refers to provides subprogram access in other thread: execution by called thread

(12) The results of a subprogram call must be available to the caller at the time those results are used. In the case of a local call the results are available when the call returns, i.e., the call is performed as a synchronous call. In the case of remote call, the caller thread is by default suspended until the execution of the subprogram completes (synchronous call). The caller thread may issue multiple concurrently executing subprogram calls and wait for their result when needed (semi-synchronous call). The `Subprogram_Call_Type` property indicates whether synchronous or semi-synchronous calls are desired.

(13) In the case of a remote call, the thread servicing the subprogram call assures that only one call at a time is serviced. In other words, it acts as a critical region for all calls to provides subprogram access features of a thread.

(14) Provides subprogram access features may be declared for processors or devices. In the case of processors they represent operating system services provided by the processor. In the case of a device, they represent services on the device that can be invoked by the application software.

Processing Requirements and Permissions

(15) The subprogram call order defines a default execution order for the subprogram calls. Alternate call orders can be modeled in an annex subclause introduced for that purpose.

(16) The legality rules require that call declarations either refer only to subprogram classifiers or to subprogram instances (subcomponents and provides/requires subprogram access). This rule can be relaxed to allow a mix of both if this is appropriate for the development process.

(17) An implementation method may support synchronous calls only or also semi-synchronous calls.

Examples

2.2 Subprogram Groups and Subprogram Group Types

(1) Subprogram groups represent subprogram libraries. As such, their dynamic semantics is similar to the dynamic semantics of a collection of subprograms.

2.3 Threads

(1) A thread models a concurrent task or an active object, i.e., a schedulable unit that can execute concurrently with other threads. Each thread represents a sequential flow of control that executes instructions within a binary image produced from source text.

(2) AADL supports an input-compute-output model of communication and execution for threads and port-based communication. The inputs received from other components are frozen at a specified point, by default the dispatch of a thread. As a result the computation performed by a thread is not affected by the arrival of new input until an explicit request for input, by default the next dispatch. Similarly, the output is made available to other components at a specified point in time, for data ports by default at completion time or thread deadline. In other words, AADL is able to support both synchronous execution and communication behavior, e.g., in the form of deterministic sampling of a control system data stream, as well as asynchronous concurrent processing.

© SAE International DRAFT

(3) Systems modeled in AADL can have operational modes (see Section 12). A thread can be active in a particular mode and inactive in another mode. As a result, a thread may transition between an active and inactive state as part of a mode switch. Only active threads can be dispatched and scheduled for execution. Threads can be dispatched periodically or as the result of explicitly modeled events that arrive at event ports, event data ports. Completion of the normal execution including error recovery will result in an event being delivered through the reserved `Complete` event out port. Completion under unrecoverable error conditions will result in an event being delivered through the reserved `Abort` and `Stop` ports.

(4) If the thread execution results in a fault that is detected, the source text may handle the error. If the error is not handled in the source text, the thread is requested to recover and prepare for the next dispatch. If an error is considered thread unrecoverable, its occurrence is reported through the reserved `Error` out event data port.

Processing Requirements and Permissions

(5) One or more AADL threads may be implemented in a single operating system thread. A thread always executes within the virtual address space of a process, i.e., the binary images making up the virtual address space must be loaded before any thread can execute in that virtual address space. Threads are dispatched, i.e., their execution is initiated periodically by the clock or by the arrival of data or events on ports, or by arrival of subprogram calls from other threads.

Semantics

(6) Thread semantics are described in terms of thread states, thread dispatching, thread scheduling and execution, and fault handling. Thread execution semantics apply once the appropriate binary images have been loaded into the respective virtual address space (see Section 5.6).

(7) Threads are dispatched at specific time instants (e.g. periodically, after a timeout_ or by the arrival of data and events, or by arrival of subprogram calls from other threads. Subprogram calls always trigger dispatches. Subsets of ports can be specified to trigger dispatches. By default, any one of the incoming **event ports** and **event data ports** triggers a dispatch.

Note: A method of processing may change this semantics through properties or annexes.

(8) Port input is frozen at dispatch time or a specified time during thread execution and made available to the thread for access in the form of a port variable (see Section 8.3). From that point on its content is not affected by new arrival of data and event for the remainder of the current execution. This assures a input-compute-output model of execution. By default, input of ports is frozen for all ports that are not candidates for thread dispatch triggering; for dispatch trigger candidates, only those port(s) actually triggering a specific dispatch is frozen. Whether input of specific ports is frozen at a dispatch and the time at which it is frozen can be explicitly specified (see Section 8.3.2).

(9) Threads may be part of modes of containing components. In that case a thread is active, i.e., eligible for dispatch and scheduling, only if the thread is part of the current mode.

(10) Threads mode subclauses define thread-internal operational modes. Threads can have property values that are different for different thread-internal modes.

(11) Every thread has a predeclared **out event port** named `Complete`. If this port is connected, then an event is raised implicitly on this port when nominal execution including recovery of a thread dispatch completes.

(12) Every thread has a predeclared **out event data port** named `Error`. If this port is connected, i.e., named as the source in a connection declaration, then an event is raised implicitly on this port when a thread unrecoverable error is detected (see Section 5.4.4 for more detail). This supports the propagation of thread unrecoverable errors as event data for fault handling by a thread.

(13) Threads may contain subprogram subcomponents that can be called from within the thread, and also by other threads if it is made accessible through a provides subprogram access declaration. Similarly, a thread can contain a subprogram group declaration, which represents an instance of a subprogram library dedicated to the thread. The subprograms within the subprogram library can be called from within the thread or by other threads if it is made accessible through a provides subprogram group access declaration. For further details about calling subprograms see Section XXX. Finally, a thread can contain data subcomponents. They represent static data owned by the thread,

© SAE International DRAFT

i.e., state that is preserved between thread dispatches. The thread has exclusive access to this data component unless it specifies it to be accessible through a provides data access declaration.

2.3.1 Thread States and Actions

(14) A thread executes a code sequence in the associated source text when dispatched and scheduled to execute. This code sequence is part of a binary image accessible in the virtual address space of the containing process. It is assumed that the process is bound to the memory that contains the binary image (see Section 5.6).

(15) A thread can be described as an automata that goes through several states. Thread state transitions under normal operation are described here and illustrated in Figure 5. Thread state transitions under fault conditions are described in Section 5.4.4

(16) The initial state is *thread halted*. When the loading of the virtual address space as declared by the enclosing process completes (see Section 5.6), a thread is *initialized* by performing an initialization code sequence in the source text. Once initialization is completed the thread enters the *suspended awaiting dispatch* state if the thread is part of the initial mode, otherwise it enters the *suspended awaiting mode* state. When a thread is in the *suspended awaiting mode* state it cannot be dispatched for execution.

(17) A thread may be declared to have modes. In this case, each mode represents a behavioral state within the execution of the thread. When a thread is dispatched it is assumed to execute in a specific mode. It may resume execution in the behavioral state (mode) in which it completed its previous dispatch execution, e.g., state reflected in a static data component, or it may execute in a specific mode based on the input received at dispatch time. A modal thread can have mode-specific property values. For example, a thread can have different worst-case execution times for different modes, each representing a different execution path through the source code. The result is a model that more accurately reflects the actual system behavior. The Behavior Model Annex Document D allows for a refined specification of thread behavior, e.g., it may explicitly specify the conditions under which the thread executes in one mode or another mode and it can represent intermediate behavioral states.

(18) When a mode transition is initiated, a thread that is part of the old mode and not part of the new mode *exits* the mode by transitioning to the *suspended awaiting mode* state after performing *thread deactivation* during the *mode change in progress* system state (see Figure 23). If the thread is periodic and its `Synchronized_Component` property is true, then its period is taken into consideration to determine the actual mode transition time (see Sections 12 and 13.6 for detailed timing semantics of a mode transition). If an aperiodic or a sporadic thread is executing a dispatch when the mode transition is initiated, its execution is handled according to the `Active_Thread_Handling_Protocol` property. The execution of a background thread is suspended through deactivation while the thread is not part of the new mode. A thread that is not part of the old mode and part of the new mode *enters* the *mode* by transitioning to the *suspended awaiting dispatch* state after performing *thread activation*.

(19) When in the *suspended awaiting dispatch* state, a thread is awaiting a dispatch request for performing the execution of a compute source text code sequence as specified by the `Compute_Entrypoint` property on the thread or on the event or event data port that triggers the dispatch. When a dispatch request is received for a thread, data, event information, and event data is made available to the thread through its port variables (see Sections 8.2 and 9.1). The thread is then handed to the scheduler to perform the computation. Upon successful completion of the computation, the thread returns to the *suspended awaiting dispatch* state. If a dispatch request is received for a thread while the thread is in the compute state, this dispatch request is handled according to the specified `Overflow_Handling_Protocol` for the event or event data port of the thread.

(20) A thread may enter the *thread halted* state, i.e., will not be available for future dispatches and will not be included in future mode switches. If re-initialization is requested for a thread in the *thread halted* state (see Section 5.6), then its virtual address space is reloaded, the processor to which the thread is bound is restarted, or the system instance is restarted.

(21) A thread may be requested to enter its *thread halted* state through a *stop* request after completing the execution of a dispatch or while not part of the active mode. In this case, the thread may execute a *finalize* entrypoint before entering the *thread halted* state. A thread may also enter the *thread halted* state immediately through an *abort* request.

(22) Any resources locked by `Get_Resource` are released (see Figure 5).

© SAE International

(23) Figure 5 presents the top-level hybrid automaton (using the notation defined in Section 1.6) to describe the dynamic semantics of a thread from the perspective of a scheduler. The hybrid automaton states complement the application modes declared for threads. Figure 7 elaborates the performing *thread computation* state of Figure 5. Figure 6 elaborates the executing *nominally* substate of Figure 7. The bold faced edge labels in Figure 5 indicate that the transitions marked by the label are coordinated across multiple hybrid automata. The scope of the labels is indicated in parentheses, i.e., interaction with the process hybrid automaton (Figure 8), with the system hybrid automaton (Figure 22) and with system wide mode switching (see Figure 23). Thread initialization is only started when the process containing the thread has been loaded as indicated by the label **loaded(process)**. The label **started(system)** is coordinated with other threads and the system hybrid automaton to transition to *System operational* only after threads have been initialized. In some systems it is desirable to initialize all threads, while in other system it is acceptable for threads to be created and initialized more dynamically, possibly even at activation and deactivation.

(24) The hybrid automata contain assertions. In a time-partitioned system these assertions will be satisfied. In other systems they will be treated as anomalous behavior.

(25) For each of the states representing a *performing thread* action such as *initialize*, *compute*, *recover*, *activate*, *deactivate*, and *finalize*, an execution entrypoint to a code sequence in the source text can be specified. Each entrypoint may refer to a different source text code sequence which contains the entrypoint, or all entrypoints of a thread may be contained in the same source text code sequence. In the latter case, the source text code sequence can determine the context of the execution through a `Dispatch_Status` runtime service call (see Section 5.4.8). The execution semantics for these entrypoints is described in Section 5.4.3.

(26) An *Initialize_Entrypoint* (enter the state `performing` *`thread initialization`* in Figure 5) is executed during system initialization and allows threads to perform application specific initialization, such as ensuring the correct initial value of its **out** and **in out** ports. A thread that has halted may be re-initialized.

(27) The *Activate_Entrypoint* (enter the state `performing` *`thread activation`* in Figure 5) and *Deactivate_Entrypoint* (enter the state `performing` *`thread activation`* in Figure 5) are executed during mode transitions and allow threads to take user-specified actions to save and restore application state for continued execution between mode switches. These entrypoints may be used to reinitialize application state due to a mode transition. Activate entrypoints can also ensure that **out** and **in out** ports contain correct values for operation in the new mode.

(28) The *Compute_Entrypoint* (enter state `performing` *`thread computation`* in Figure 5) represents the code sequence to be executed on every thread dispatch. Each provides subprogram access feature represents a separate compute entrypoint of the thread. Remote subprogram calls are thread dispatches to the respective entrypoint. Event ports and event data ports can have port specific compute entrypoints to be executed when the corresponding event or event data dispatches a thread.

(29) A *Recover_Entrypoint* (enter the state `executing` *`recovery`* in Figure 7) is executed when a fault in the execution of a thread requires recovery activity to continue execution. This entrypoint allows the thread to perform fault recovery actions (for a detailed description see Section 5.4.4).

(30) A *Finalize_Entrypoint* (enter the state `performing` *`thread finalize`* in Figure 5) is executed when a thread is asked to terminate as part of a process unload or process stop.

(31) If no value is specified for any of the entrypoints, then there is no invocation at all.

© SAE International DRAFT

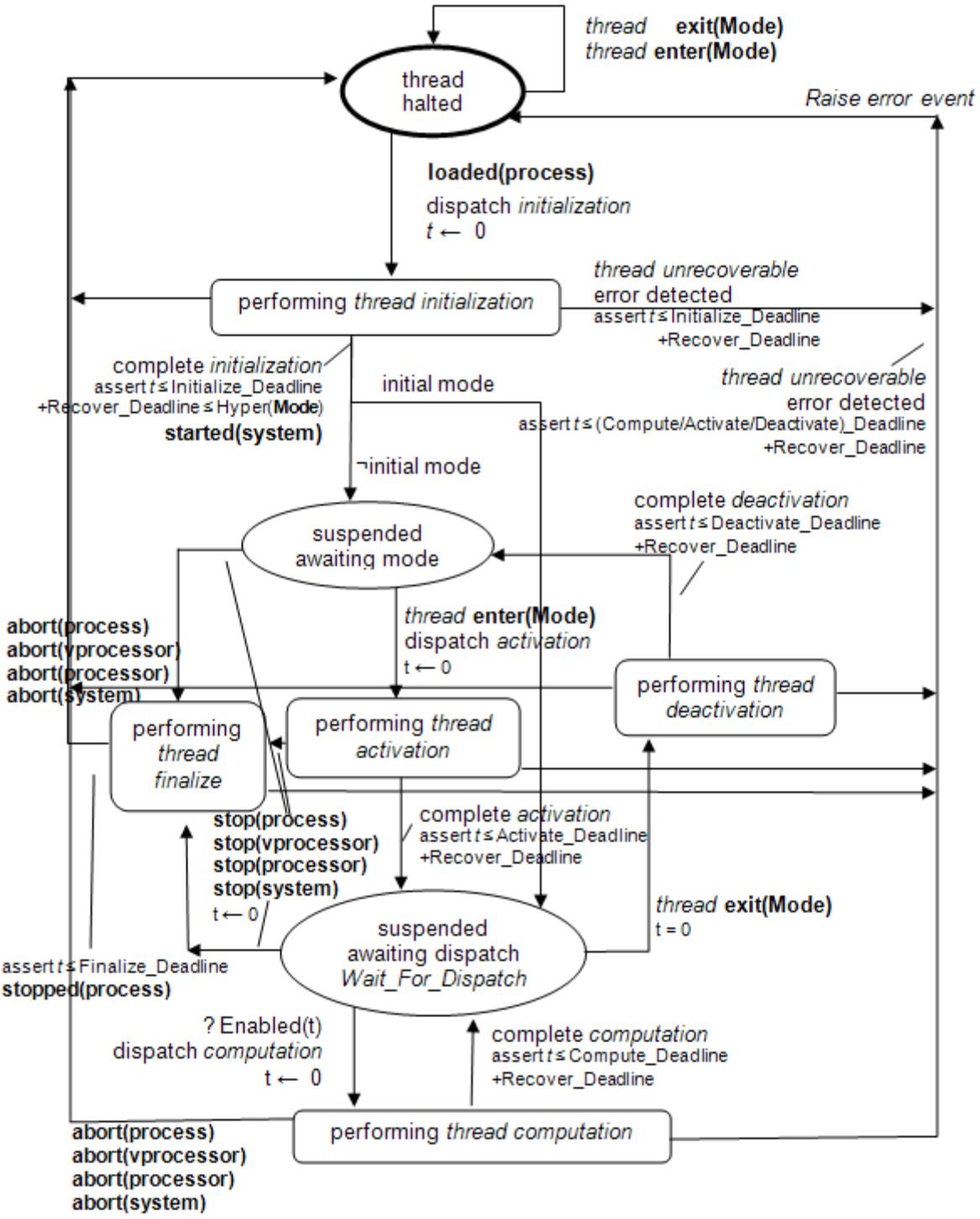

Figure 1 − Thread States and Actions

2.3.2 Thread Dispatching

(32) Threads are dispatched periodically determined by a clock or by the arrival of events, event data, of calls to provides subprogram access. By default any event port or event data port can trigger a dispatch. In that case, only the input of the port triggering the dispatch and any data port is available to the application program.

(33) A thread may have a `Dispatch_Trigger` property to specify a subset of event, data, or event data ports that can trigger a thread dispatch. In this case, arrival of events or event data on any of the listed ports can trigger the dispatch.

(34) The default disjunction of ports triggering a dispatch can be overwritten by a logical condition on the ports expressed by a annex subclauses of the Behavior Annex notation (see Annex Document D).

(35) For periodic threads arrival of events or event data will not result in a dispatch. Events and event data are queued in their incoming port and are accessible to the application code of the thread. Periodic thread dispatches are solely determined by the clock according to the time interval specified through the `Period` property value.

(36) The `Dispatch_Protocol` property of a thread determines the characteristics of dispatch requests to the thread. This is modeled in the hybrid automaton in Figure 5 by the `Enabled(t)` function as the `Wait_For_Dispatch` invariant. The `Enabled` function determines when a transition from `Wait_For_Dispatch` to performing thread computation will occur. The `Wait_For_Dispatch` invariant captures the condition under which the `Enabled` function is evaluated. The consequence of a dispatch is the execution of the entrypoint source text code sequence at its *current execution* position. This position is set to the first step in the code sequence and reset upon completion (see Section 5.4.3).

(37) For a thread whose dispatch protocol is `periodic`, a dispatch request is issued at time intervals of the specified `Period` property value. The `Enabled` function is $t = Period$. The `Wait_For_Dispatch` invariant is $t \leq Period \wedge \delta t = 1$. The dispatch occurs at $t = Period$. The `Compute_Entrypoint` of the thread is called.

(38) Periodic threads can have a `Dispatch_Offset` property value. In this case the dispatch time is offset from the period by the specified amount. This allows two periodic threads with the same period to be aligned, where the first thread has a pre-period deadline, and the second thread has a dispatch offset greater than the deadline of the first thread. This is a static alignment of thread execution order within a frame, while the immediate data connection achieves the same by dynamically aligning completion time of the first thread and the start of execution of the second thread (see Section 9.2.5).

(39) For threads whose dispatch protocol is `aperiodic`, `sporadic`, `timed`, or `hybrid`, a dispatch request is the result of an event or event data arriving at an event or event data port of the thread, or a remote subprogram call arriving at a provides subprogram access feature of the thread. This *dispatch trigger condition* is determined as follows:

- Arrival of an event or event data on any incoming event, or event data port, or arrival of any subprogram call request on a provides subprogram access feature. In other words, it is a disjunction of all incoming features.
- By arrival on a subset of incoming features (port, subprogram access). This subset can be specified through the `Dispatch_Trigger` value of the thread.
- By a user-defined logical condition on the incoming features that can trigger the dispatch expressed through an annex subclause expressed in the Behavior Annex sublanguage notation (see Annex Document D).

(40) For a thread whose dispatch protocol is `aperiodic`, a dispatch request is the result of an event or event data arriving at an event or event data port of the thread, or a remote subprogram call arriving at a provides subprogram access feature of the thread. There is no constraint on the inter-arrival time of events, event data or remote subprogram calls. The dispatch actually occurs immediately when a dispatch request arrives in the form of an event at an event port with an empty queue, or if an event is already queued when a dispatch execution completes, or a remote subprogram call arrives. The `Enabled` function by default has the value true if there exists a port or provides subprogram access (`p`) in the set of features that can trigger a dispatch (E) with a non-empty queue, i.e., $\exists$ `p in E:` $p \neq \emptyset$. This evaluation function may be redefined by the Behavior Annex (see Annex Document D). The `Wait_For_Dispatch` invariant is that no event, event data, or subprogram call is queued, i.e., $\forall$ `p in E:` $p = \emptyset$. The `Compute_Entrypoint` of the port triggering the dispatch, or if not present that of the thread, is called.

(41) If multiple ports are involved in triggering the dispatch the `Compute_Entrypoint` of the thread is called. The list of ports actually satisfying the dispatch trigger condition that results in the dispatch is available to the source text as output parameter of the `Await_Dispatch` service call (see Section 5.4.8).

© SAE International DRAFT

(42) For a thread whose dispatch protocol is sporadic, a dispatch request is the result of an event or event data arriving at an event or event data port of the thread, or a remote subprogram call arriving at a provides subprogram access feature of the thread. The time interval between successive dispatch requests will never be less than the associated Period property value. The Enabled function is $t \geq Period \wedge \exists p \text{ in } E: p \neq \emptyset$. The Wait_For_Dispatch invariant is $t < Period \vee (t > Period \wedge \forall p \text{ in } E: p = \emptyset)$. The dispatch actually occurs when the time condition on the dispatch transition is true and a dispatch request arrives in the form of an event at an event port with an empty queue, or an event is already queued when the time condition becomes true, or a remote subprogram call arrives when the time condition is true. The Compute_Entrypoint of the port triggering the dispatch, or if not present that of the thread, is called.

(43) For a thread whose dispatch protocol is timed, a dispatch request is the result of an event, event data, or remote subprogram arrival, or it occurs by an amount of time specified by the Period property since the last dispatch.. In other words, the Period represents a time-out value that ensure a dispatch occurs after a given amount of time if no events, event data, or remote subprogram calls have arrived or are queued. The Enabled function by default has the value true if there exists a port or provides subprogram access (p) in the set of features that can trigger a dispatch (E) with an event, event data, or call in its queue, or time equal to the Period has expired since the last dispatch, i.e., $\exists p \text{ in } E: p \neq \emptyset \vee t = Period$. t is reset to zero at each dispatch. This evaluation function may be redefined by the Behavior Annex (see Annex Document D). The Wait_For_Dispatch invariant is that no event, event data, or call is queued, i.e., $\forall p \text{ in } E: p = \emptyset \wedge t < Period$. The Compute_Entrypoint of the port triggering the dispatch, or if not present that of the thread, is called. If a timeout occurs, i.e., the dispatch is triggered at the end of the period, the Recover_Entrypoint is called.

(44) A thread whose dispatch protocol is hybrid, combines both aperiodic and periodic dispatch behavior in the same thread. A dispatch request is the result of an event, event data, or remote subprogram call arrival, as well as periodic dispatch requests at a time interval specified by the Period property value. The Enabled function is $t = Period \vee \exists p \text{ in } E: p \neq \emptyset$. t is reset to zero at each periodic dispatch. The evaluation function for events, event data, or subprogram calls may be redefined by the Behavior Annex. The Wait_For_Dispatch invariant is that no event, event data, call, or periodic dispatch is queued and the period has not expired, i.e., $\forall p \text{ in } E: p = \emptyset \wedge t \leq Period$. The Compute_Entrypoint of the port triggering the dispatch, or if not present that of the thread, is called.

(45) If several events or event data occur logically simultaneously and are routed to the same port of an aperiodic, sporadic, timed, or hybrid thread, the order of arrival for the purpose of event handling according the above rules is implementation-dependent. If several events or event data occur logically simultaneously and are routed to the different ports of the same aperiodic, sporadic, timed, or hybrid thread, the order of event handling is determined by the Urgency property associated with the ports.

(46) For a thread whose dispatch protocol is background, the thread is dispatched upon completion of its initialization entrypoint execution the first time it is active in a mode. The Enabled function is true. The Wait_For_Dispatch invariant is $t = 0$. The dispatch occurs immediately. If the Dispatch_Trigger property is set, then its execution is initiated through the arrival of an event or event data on one of those ports. In that case, the Enabled function is true for any Dispatch port $p \in Dispatch\_Trigger : p \neq \emptyset$.

© SAE International DRAFT

(47) The different dispatch protocols can be summarized as follows:

Dispatch protocol	Dispatch condition
Periodic	Every period triggered by the dispatcher of the runtime system
Aperiodic	Triggered by the arrival of events, event data, subprogram calls
Sporadic	Triggered by the arrival of events, event data, subprogram calls with a minimum time difference of the specified period between dispatches
Timed	Triggered by the arrival of events, event data, subprogram calls with a timeout at the specified period.
Hybrid	Triggered by the arrival of events, event data, subprogram calls, as well as every period triggered by the dispatcher.

(48) Note that background threads do not have their current execution position reset on a mode switch. In other words, the background thread will resume execution from where it was previously suspended due to a mode switch.

(49) Note that background threads are suspended when not active in the current mode. If they have access to shared data components, they may have locked the resource at the time of suspension and potentially cause deadlock if active threads also share access to the same data component.

(50) A background thread is scheduled to execute such that all other threads' timing requirements are met. If more than one background thread is dispatched, the processor's scheduling protocol determines how such background threads are scheduled. For example, a FIFO protocol for background threads means that one background thread at a time is executed, while fair share means that all background threads will make progress in their execution.

(51) The `Overflow_Handling_Protocol` property for event or event data ports specifies the action to take when events arrive too frequently. These events are ignored, queued, or are treated as an error. The error treatment causes the currently active dispatch to be aborted, allowing it to clean up through the `Recover_Entrypoint` and then be redispatched. For more details on port queuing see section 8.3.3.

Examples

```
thread Prime_Reporter
features
  Received_Prime : in event data port Base_Types::Integer;
properties
  Dispatch_Protocol => Timed;
end Prime_Reporter;

thread Prime_Reporter_One extends Prime_Reporter
features
  Received_Prime : refined to in event data port Base_Types::Integer
    {Compute_Entrypoint_Source_Text => "Primes.On_Received_Prime_One";};
    -- function called on message-based dispatch
properties
  Period => 9 Sec; -- timeout period
  Priority => 45;
```

© SAE International DRAFT

```
  Compute_Entrypoint_Source_Text => "Primes.Report_One";
  -- function called in case of timeout
end Prime_Reporter_One;
```

2.3.3 Thread Scheduling and Execution

(52) When a thread action is *performing thread computation* (see Figure 5), the execution of the thread's entrypoint source text code sequence is managed by a scheduler. This scheduler coordinates all thread executions on one processor as well as concurrent access to shared resources. While performing the execution of an entrypoint the thread can be *executing nominally* or *executing recovery* (see Figure 7). While executing an entrypoint a thread can be in one of five substates: ready, running, awaiting resource, awaiting return, and awaiting resume (see Figure 6).

(53) A thread initially enters the *ready* state. A scheduler selects one thread from the set of threads in the ready state to run on one processor according to a specified scheduling protocol. It ensures that only one thread is in the *running* state on a particular processor. If no thread is in the ready state, the processor is idle until a thread enters the ready state. A thread will remain in the running state until it completes execution of the dispatch, until a thread entering the ready state preempts it if the specified scheduling protocol prescribes preemption, until it blocks on a shared resource, or until an error occurs. In the case of completion, the thread transitions to the suspended *awaiting dispatch* state, ready to service another dispatch request. In the case of preemption, the thread returns to the ready state. In the case of resource blocking, it transitions to the *awaiting resource* state.

(54) Shared data is accessed in a critical region. Resource blocking can occur when a thread attempts enter a critical region while another thread is already in this critical region. In this case the thread enters the *Awaiting resource* state. A `Concurrency_Control_Protocol` property value associated with the shared data component determines the particular concurrency control mechanism to be used (see Section 5.1). The `Get_Resource` and `Release_Resource` service calls are provided to indicate the entry and exit of critical regions (see Section 5.1.1). When a thread completes execution it is assumed that all critical regions have been exited, i.e., access control to shared data has been released. Otherwise, the execution of the thread is considered erroneous.

(55) Subprogram calls to remote subprograms are synchronous or semi-synchronous. In the synchronous case, a thread in the running state enters the *awaiting return* state when performing a call to a subprogram whose service is performed by a subprogram in another thread. The service request for the execution of the subprogram is transferred to the remote subprogram request queue of a thread as specified by the `Actual_Subprogram_Call` property that specifies the binding of the subprogram call to a subprogram in another thread. When the thread executing the corresponding remote subprogram completes and the result is available to the caller, the thread with the calling subprogram transitions to the ready state. In the semi-synchronous case, the calling thread continues to execute concurrently until it awaits the result of the call (see service call `Await_Result` in Section 5.4.8).

(56) A background thread may be temporarily suspended by a mode switch in which the thread is not part of the new mode, as indicated by the **exit(Mode)** in Figure 6. In this case, the thread transitions to the *awaiting mode_entry* state. If the thread was in a critical region, it will be suspended once it releases all resources on exit of the critical region. A background thread resumes execution when it becomes part of the current mode again in a later mode switch. It then transitions from the *awaiting_mode_entry* state into the *ready* state.

(57) Execution of any of these entrypoints is characterized by actual execution time (*c*) and by elapsed time (*t*). Actual execution time is the time accumulating while a thread actually runs on a processor. Elapsed time is the time accumulating as real time since the arrival of the dispatch request. Accumulation of time for *c* and *t* is indicated by their first derivatives δc and δt. A derivative value of 1 indicates time accumulation and a value of 0 indicates no accumulation. Figure 6 shows the derivative values for each of the scheduling states. A thread accumulates actual execution time only while it is in the running state. The processor time, if any, required to switch a thread between the running state and any of the other states, which is specified in the `Thread_Swap_Execution_Time` property of the processor, is not accounted for in the `Compute_Execution_Time` property, but must be accounted for by an analysis tool.

(58) The execution time and elapsed time for each of the entrypoints are constrained by the entrypoint-specific `<entrypoint>_Execution_Time` and entrypoint-specific `<entrypoint>_Deadline` properties specified for

DRAFT © SAE International

the thread. If no entrypoint specific execution time or deadline is specified, those of the containing thread apply. There are three timing constraints:

- Actual execution time, *c*, will not exceed the maximum entrypoint-specific execution time.
- Upon execution completion the actual execution time, *c*, will have reached at least the minimum entrypoint-specific execution time.
- Elapsed time, *t*, will not exceed the entrypoint-specific deadline.

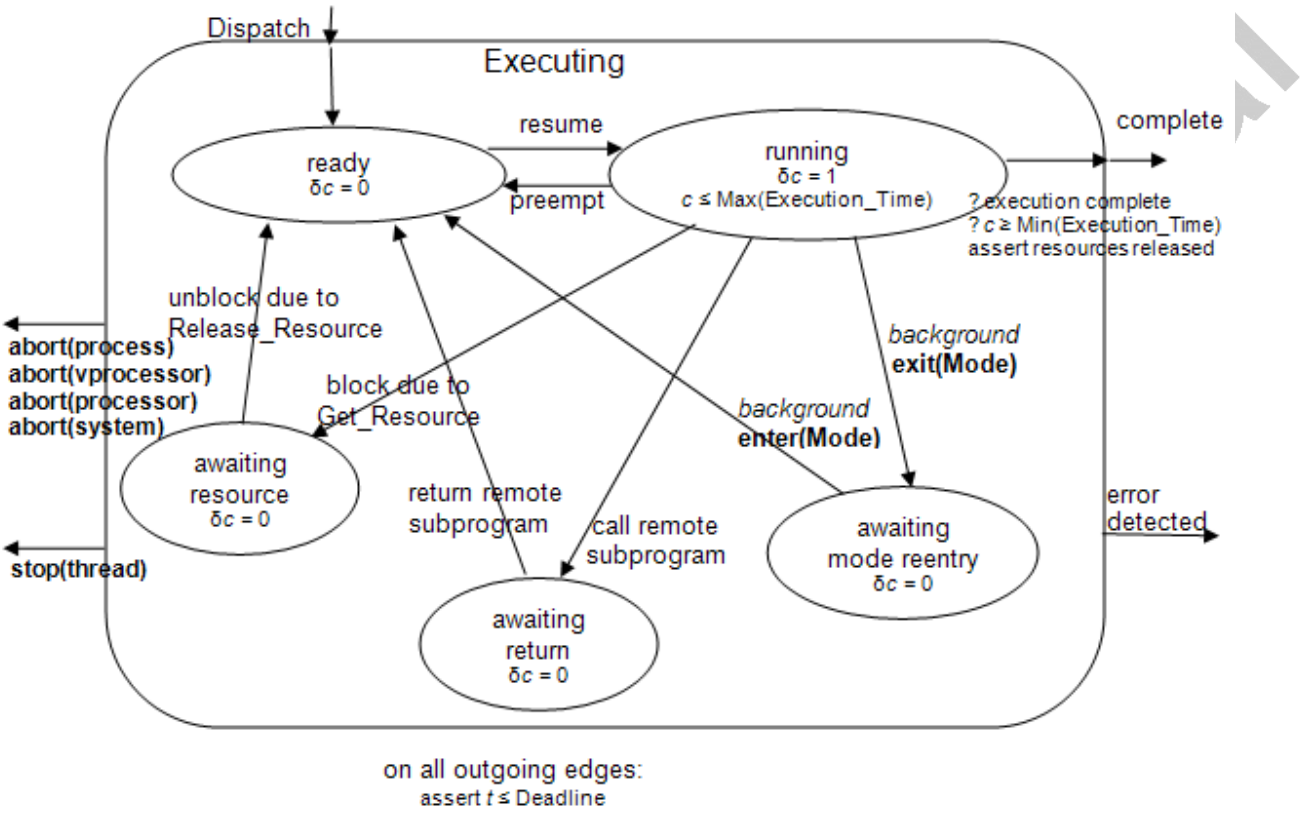

Figure 2 − Thread Scheduling and Execution States

(59) Execution of a thread is considered anomalous when the timing constraints are violated. Each timing constraint may be enforced and reported as an error at the time, or it may be detected after the violation has occurred and reported at that time. The implementor of a runtime system must document how it handles timing constraints.

2.3.4 Execution Fault Handling

(60) A fault is defined to be an anomalous undesired change in thread execution behavior, possibly resulting from an anomalous undesired change in data being accessed by that thread or from violation of a compute time or deadline constraint. An error is a fault that is detected during the execution of a thread. Detectable errors are classified as *thread recoverable* errors, or *thread unrecoverable* errors.

(61) A *thread recoverable* error may be handled as part of normal execution by that thread, e.g., by exception handlers programmed in the source text of the application. The exception handler may propagate the error to an external handler by sending an event or event data through a port.

(62) If the thread recoverable error is not handled by the application, the thread affected by the error is given a chance to recover through the invocation of the thread's recover entrypoint. The recover entrypoint source text sequence has the opportunity to update the thread's application state. The recover entrypoint is assumed to have access to an error code through a runtime service call `Get_Error_Code`. The recover entrypoint may report the fact that it performed recovery through a user–defined port. Upon completion of the recover entrypoint execution, the performance of the thread's dispatch is considered complete. In the case of performing thread computation, this

© SAE

means that the thread transitions to the suspended await dispatch state (see Figure 5), ready to perform additional dispatches. Concurrency control on any shared resources must be released. If the recover entrypoint is unable to recover the error becomes a *thread unrecoverable* error. This thread-level fault handling in terms of thread scheduling states is illustrated in Figure 7.

(63) A thread recoverable error may occur during the execution of a remote subprogram call. In this case, the thread servicing the remote call is given a chance to recover as well as the thread that made the call.

(64) In the presence of a thread recoverable error, the maximum interval of time between the dispatch of a thread and its returning to the suspended awaiting dispatch state is the sum of the thread's compute deadline and its recover deadline. The maximum execution time consumed is the sum of the compute execution time and the recover execution time. In the case when an error is encountered during recovery, the same numbers apply.

(65) *Thread unrecoverable* errors are reported as event data through the `Error` port of the thread, where they can be communicated to a separate error handling thread for further analysis and recovery actions.

(66) A thread unrecoverable error causes the execution of a thread to be terminated prematurely without undergoing recovery. The thread unrecoverable error is reported as an error event through the predeclared `Error` event data port, if that port is connected. If this implicit error port is not connected, the error is not propagated and other parts of the system will have to recognize the fault through their own observations. In the case of a thread unrecoverable error, the maximum interval between the dispatch of the thread and its returning to the suspended awaiting dispatch state is the compute deadline, and the maximum execution time consumed is the compute execution time.

(67) For errors detected by the runtime system, error details are recorded in the data portion of the event as specified by the implementation. For errors detected by the source text, the application can choose its encoding of error detail and can raise an event in the source text. If the propagated error will be used to directly dispatch another thread or trigger a mode change, only an event needs to be raised. If the recovery action requires interpretation external to the raising thread, then an event with data must be raised. The receiving thread that is triggered by the event with data can interpret the error data portion and raise events that trigger the intended mode transition.

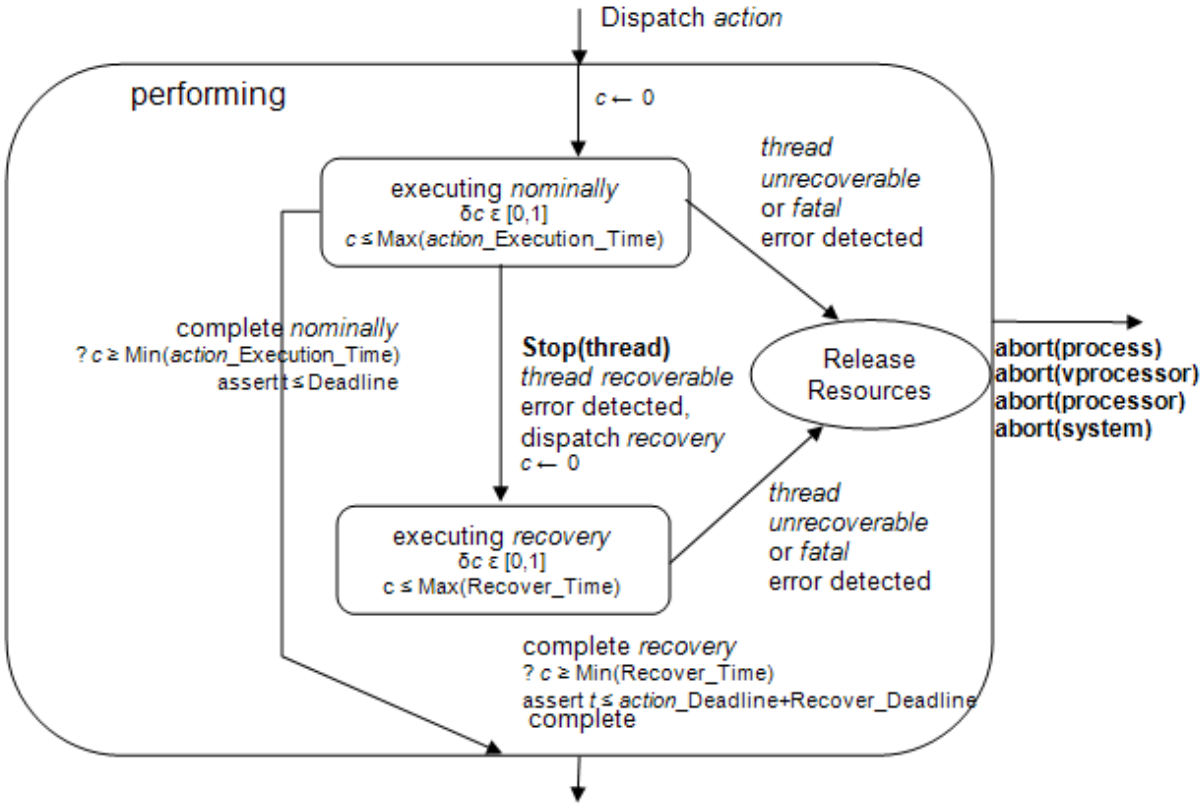

Figure 3 − Performing Thread Execution with Recovery

(68) A timing fault during initialize, compute, activation, and deactivation entrypoint executions is considered to be a thread recoverable error. A timing fault during recover entrypoint execution is considered to be a thread unrecoverable error.

(69) If any error is encountered while a thread is executing a recover entrypoint, it is treated as a thread unrecoverable error. In other words, an error during recovery must not cause the thread to recursively re-enter the executing recovery state.

(70) If a fault is encountered by the application itself, it may explicitly raise an error through a `Raise_Error` service call on the `Error` port with the error class as parameter. This service call may be performed in the source text of any entrypoint. In the case of recover entrypoints, the error class must be *thread unrecoverable*.

(71) Faults may also occur in the execution platform. They may be detected by the execution platform components themselves and reported through an event or event data port, as defined by the execution platform component. They may go undetected until an application component such as a health monitoring thread detects missing health pulse events, or a periodic thread detects missing input. Once detected, such errors can be handled locally or reported as event data.

2.3.5 Thread Internal Modes and Mode Transitions

(72) A thread can have modes declared inside its type or implementation. They represent thread-internal execution paths and allow mode-specific property values to be associated with the thread. For example, the thread can have different execution times under different modes. Application source text (programming code) actually executes branch and merge points in various places of its code sequence and branches based on state values or based on input. In terms of the mode abstraction this means that the (mode) state at time of dispatch may affect the branch condition, the input to the thread execution may affect the branch condition, or a combination, or a change in the state value is the result of computation based on the previous value and/or input.

(73) A mode transition of a thread-internal mode may be implicit in that it is determined by the application source code of the thread. This source code may follow and execution sequence based on the content of thread input or by the value of a static data variable. In the former case, the current mode is determined at the time the thread input is determined, by default thread dispatch time. In the latter case, the value of the static data determines the current mode at the next dispatch. The effect is a possible change in mode-specific property values to reflect a change in source text internal execution behavior, e.g., a change in worst-case execution time, and in the entrypoint or call sequence to be executed at the next dispatch.

(74) Change of a thread-internal mode may be explicitly modeled by declaring a mode transition that names an incoming thread port, an event raised within a thread (declared as **self**.eventname), or names a subprogram call with an outgoing event. Change of a thread-internal mode may also be modeled through the Behavior Model Annex Document D. In this case, mode transitions are tracked by the runtime system to determine the most recent current mode for the next dispatch of a thread.

(75) If a higher fidelity behavioral model is desired, the Behavior Annex (Annex Document D), which uses the AADL modes as the initial set of states, can be used for more complex behavioral specifications. The final authority of the actual behavior of the source text is the program code itself.

2.3.6 System Synchronization Requirements

(76) An application system may consist of multiple threads. Each thread has its own hybrid automaton state with its own *c* and *t* variables. This results in a set of concurrent hybrid automata. In the concurrent hybrid automata model for the complete system, `ST` is a single real-valued variable shared by all threads that is never reset and whose rate is 1 in all states. `ST` is called the *reference timeline*.

(77) A set of periodic threads are said to be logically dispatched simultaneously at global real time *ST* if the order in which all exchanges of control and data at that dispatch event are identical to the order that would occur if those dispatches were exactly dispatched simultaneously in true and perfect real time. The *hyperperiod h* of a set of periodic threads is the next time *ST+h* at which they are logically dispatched simultaneously. The hyperperiod is the least common multiple of the periodic thread periods.

© SAE International DRAFT

(78) An application system is said to be synchronized if the dispatch of all periodic threads contained in that application system occurs logically simultaneously at intervals of their hyperperiod. In a globally synchronous system ST is a global *reference time*, i.e., a single real-valued variable representing a global clock. It represents a single global *synchronization domain*.

(79) Within a synchronization domain, perfect synchronization may not occur in an actual system. There may always be clock error bounds in a distributed clock, and jitter in exactly when events (like a dispatch) would occur even with perfect clock interrupts due to things like non-preemptive blocking times (during which clock interrupts might be masked briefly). Within a synchronization domain, it is the responsibility of each physical implementation to take these imperfections into account when providing the synchronization domain for programmers (e.g., make sure the message transmission schedule includes enough margin for the message to arrive at the destination by the time it is needed, taking into account these various effects in the particular implementation).

2.3.7 Asynchronous Systems

(80) In a globally asynchronous system there are multiple *reference times*, i.e., multiple variables ST_j. They represent different *synchronization domains*. Any time related coordination and communication between threads, processors, and devices across different synchronization domains must take into account differences in the *reference time* of each of those synchronization domains.

(81) Reference times in the form of a clock or time domain can be represented by instances of a user-defined abstract, processor, or device type. Characteristics of this reference time component, such as clock drift rate and maximum clock drift can be specified as properties of these instances. Processors, devices, buses, memory, virtual buses, virtual processors, and systems can be assigned different reference times through the Reference_Time property. Similarly, application components can be assigned reference times to represent the fact that they may read the time, e.g., to timestamp data.The reference time for thread execution is determined by the reference time of the processor on which the thread executes. The reference time of communication between threads, devices, and processors is determined by the reference time of the source and destination, and the reference time of any execution platform component involved in the communication if it is time-driven.

(82) The reference time for thread execution is determined by the reference time of the processor on which the thread executes. The reference time of communication between threads, devices, and processors is determined by the reference time of the source and destination, and the reference time of any execution platform component involved in the communication if it is time-driven.

(83) Message-passing semantics of communication and thread execution is represented by aperiodic threads whose dispatch is triggered by arrival of messages and message may be queued in the event data port. This communication paradigm is insensitive to time, thus, not affected by multiple synchronization domains.

(84) Data-stream semantics of communication and thread execution are represented by periodic threads and data ports. In this case the sampling of the input is sensitive to a common reference time between the source and the destination thread if the connections are immediate and delayed to ensure deterministic communication. Deterministic communication minimizes latency jitter, while non-deterministic communication can result in latency jitter in units of the sampling rate, the latter often leading to instability of latency sensitive applications such as control systems. In the case of sampling data port connections the non-deterministic nature of sampling accommodates different reference times. Similarly, a periodic thread may non-deterministically sample event ports and event data ports, e.g., a health monitor sampling an alarm queue.

(85) The Allowed_Connection_Type property of a bus specifies the types of connections supported by a bus. Buses that connect processors with different reference times may exclude immediate and delayed connections from their support if determinism cannot be guaranteed through a protocol.

(86) Mode switching requires time-sensitive coordination of deactivation and activation of threads and connections. There is the time ordering of events that request mode switching, and the coordination of switching modes in multiple modal subsystems as part of a single mode switch. Timed coordination can be guaranteed within one synchronization domain and may be feasible across synchronization domains with bounded time drift through appropriate protocols.

(87) Solutions have been devised to address this issue.

© SAE International DRAFT

- ARINC653: Thread execution and communication within a partition is assumed to be within the same synchronization domain, cross-partition communication is assumed to be message-based or (phase-)delayed for sampling ports. This assures that placement of partitions on different processors or at different parts of the timeline within one processor does not affect the timing. However, this delayed communication places a synchronicity requirement on those partitions that communicate with each other.
- Globally Asynchronous Locally Synchronous (GALS): This model reflects the fact that some systems cannot be globally synchronized, e.g., integrated modular avionics (IMA) system may consist of a collection of ARINC653 subsystems and interact via an ARINC664 network. In this case the burden is placed on the application system to deal with synchronicity within subsystems and asynchronicity across subsystems. This can be reflected in AADL by multiple synchronization domains and the requirement that data port connections across synchronization domains are sampled connections.
- Time Triggered Architecture (TTA): In this model a central communication medium provides a statically allocated time-division protocol and acts as a global reference time. Either part of the protocol provides reference time ticks to subsystems. Execution of subsystems can be aligned with the arrival of data at assigned time slots in the communication protocol to assure deterministic communication of data streams.
- Physically Asynchronous Logically Synchronous (PALS): In this model a logical protocol or application layer provides coordination of time-sensitive events across asynchronous subsystems. For example, the system may periodically re-synchronize clocks, thus, bound clock drift. This clock drift bound may be accommodated by appropriate time slack the same way jitter in a synchronous system is accommodated. Similarly, hand-shaking protocols may be used to coordinate less frequently occurring synchronization events, e.g., globally synchronous mode switching if required.

2.3.8 Runtime Support For Threads

(88) A standard set of runtime services are provided. The application program interface for these services is defined in the applicable source language annex of this standard. They are callable from within the source text. The following subprograms may be explicitly called by application source code, or they may be called by an AADL runtime system that is generated from an AADL model.

(89) The `Await_Dispatch` runtime service is called to suspend the thread execution at completion of its dispatch execution. It is the point at which the next dispatch resumes. The service call takes several parameters. It takes a `DispatchPort` list and an optional trigger condition function to identify the ports and the condition under which the dispatch is triggered. If the condition function is not present any of the ports in the list can trigger the dispatch. It takes a `DispatchedPort` as out parameter to return the port(s) that triggered the dispatch. It takes `OutputPorts` and `InputPorts` as port lists. `OutputPorts`, if present, identifies the set of ports whose sending is initiated at completion of execution, equivalent to an implicit `Send_Output` service call. `InputPorts`, if present, identifies the set of ports whose content is received at the next dispatch, equivalent to an implicit `Receive_Input` service call.

```
subprogram  Await_Dispatch
features
  -- List of ports whose output is sent at completion/deadline
  OutputPorts: in parameter <implementation-defined port list>;
  -- List of ports that can trigger a dispatch
  DispatchPorts: in parameter <implementation-defined port list>;
  -- list of ports that did trigger a dispatch
  DispatchedPort: out parameter < implementation-defined port list>;
  -- optional function as dispatch guard, takes port list as parameter
  DispatchConditionFunction: requires subprogram access;
  -- List of ports whose input is received at dispatch
  InputPorts: in parameter <implementation-defined port list>;
end Await_Dispatch;
```

© SAE International DRAFT

(90) A `Raise_Error` runtime service shall be provided that allows a thread to explicitly raise a thread recoverable or thread unrecoverable error. `Raise_Error` takes an error type identifier as parameter.

```
subprogram  Raise_Error
features
  errorID: in parameter <implementation-defined error type>;
end Raise_Error;
```

(91) A `Get_Error_Code` runtime service shall be provided that allows a recover entrypoint to determine the type of error that caused the entrypoint to be invoked.

```
subprogram Get_Error_Code
features
  errorID: out parameter <implementation-defined error type>;
end Get_Error_Code;
```

(92) Subprograms have event ports but do not have an error port. If a `Raise_Error` is called, it is passed to the error port of the enclosing thread. If a `Raise_Error` is called by a remotely called subprogram, the error is passed to the error port of the thread executing the remotely called subprogram. The `Raise_Error` method is permitted to have an error identification as parameter value. This error identification can be passed through the error port as the data value, since the error port is defined as event data port.

(93) A `Await_Result` runtime service shall be provided that allows an application to wait for the result of a semi-synchronous subprogram call.

```
subprogram Await_Result
features
  CallID: in parameter <implementation-defined call ID>;
end Await_Result;
```

Processing Requirements and Permissions

(94) Multiple models of implementation are permitted for the dispatching of threads.

- One such model is that a runtime executive contains the logic reflected in Figure 5 and calls on the different entrypoints associated with a thread. This model naturally supports source text in a higher level domain language.
- An alternative model is that the code in the source text includes a code pattern that reflects the logic of Figure 5 through explicitly programmed calls to the standard `Await_Dispatch` runtime service, including a repeated call (while loop) to reflect repeated dispatch of the compute entrypoint code sequence.

(95) Multiple models of implementation are permitted for the implementation of thread entrypoints.

- One such model is that each entrypoint is a possibly separate function in the source text that is called by the runtime executive. In this case, the logic to determine the context of an error is included in the runtime system.
- A second model of implementation is that a single function in the source text is called for all entrypoints. This function then invokes an implementer-provided `Dispatch_Status` runtime service call to identify the context of the call and to branch to the appropriate code sequence. This alternative is typically used in conjunction with the source text implementation of the dispatch loop for the compute entrypoint execution.

(96) A method of implementing a system is permitted to choose how executing threads will be scheduled. A method of implementation is required to verify to the required level of assurance that the resulting schedule satisfies the period and deadline properties. That is, a method of implementing a system should schedule all threads so that the specified timing constraints are always satisfied.

DRAFT © SAE International

(97) The use of the term “preempt” to name a scheduling state transition in Figure 6 does not imply that preemptive scheduling disciplines must be used; non-preemptive disciplines are permitted.

(98) Execution times associated with transitions between thread scheduling states, for example context swap times (specified as properties of the hosting processor), are not billed to the thread’s actual execution time, i.e., are not reflected in the `Compute Execution Time` property value. However, these times must be included in a detailed schedulability model for a system. These times must either be apportioned to individual threads, or to anonymous threads that are introduced into the schedulability model to account for these overheads. A method of processing specifications is permitted to use larger compute time values than those specified for a thread in order to account for these overheads when constructing or analyzing a system.

(99) A method of implementing a system must support the periodic dispatch protocol. A method of implementation may support only a subset of the other standard dispatch protocols. A method of implementation may support additional dispatch protocols not defined in this standard.

(100) A method of implementing a system may perform loading and initialization activities prior to the start of system operation. For example, binary images of processes and initial data values may be loaded by permanently storing them in programmable memory prior to system operation.

(101) A method of implementing a system must specify the set of errors that may be detected at runtime. This set must be exhaustively and exclusively divided into those errors that are thread recoverable or thread unrecoverable, and those that are exceptions to be handled by language constructs defined in the applicable programming language standard. The set of errors classified as source language exceptions may be a subset of the exceptions defined in the applicable source language standard. That is, a method of implementation may dictate that a language-defined exceptional condition should not cause a runtime source language exception but instead immediately result in an error. For each error that is treated as a source language exception, if the source text associated with that thread fails to properly catch and handle that exception, a method of implementation must specify whether such unhandled exceptions are thread recoverable or thread unrecoverable errors.

(102) A consequence of the above permissions is that a method of implementing a system may classify all errors as thread unrecoverable, and may not provide an executing recovery scheduling state and transitions to and from it.

(103) A method of implementing a system may enforce, at runtime, a minimum time interval between dispatches of sporadic threads. A method of implementing a system may enforce, at runtime, the minimum and maximum specified execution times. A method of implementing a system may detect at runtime timing violations.

(104) A method of implementing a system may support handling of errors that are detected while a thread is in the suspended, ready, or blocked state. For example, a method of implementation may detect event arrivals for a sporadic thread that violate the specified period. Such errors are to be kept pending until the thread enters the executing state, at which instant the errors are raised for that thread and cause it to immediately enter the recover state.

(105) If alternative thread scheduling semantics are used, a thread unrecoverable error that occurs in the perform thread initialization state may result in a transition to the perform thread recovery state and thence to the suspended awaiting mode state, rather than to the thread halted state. The deadline for this sequence is the sum of the initialization deadline and the recovery deadline.

(106) If alternative thread scheduling semantics are used, a method of implementation may prematurely terminate threads when a system mode change occurs that does not contain them, instead of entering suspended awaiting mode. Any blocking resources acquired by the thread must be released.

(107) If alternative thread scheduling semantics are used, the load deadline and initialize deadline may be greater than the period for a thread. In this case, dispatches of periodic threads shall not occur at any dispatch time prior to the initialization deadline for that periodic thread.

© SAE International DRAFT

(108) This standard does not specify which thread or threads perform virtual address space loading. This may be a thread in the runtime system or one of the application threads.

NOTE: The deadline of a calling thread will impose an end-to-end deadline on all activities performed by or on behalf of that thread, including the time required to perform any remote subprogram calls made by that thread. The deadline property of a remotely called subprogram may be useful for scheduling methods that assign intermediate deadlines in the course of producing an overall end-to-end system schedule.

2.4 Thread Groups

(1) A thread group represents an organizational component to logically group threads contained in processes. As such, they could be considered as a collection of threads with regards to their dynamic behavior.

2.5 Processes

(1) A process represents a virtual address space, i.e., it represents a space partition unit whose boundaries are enforced at runtime.

Semantics

(2) Every process has its own virtual address space. This address space provides access to source code and data associated with the process and all its contained components. This address space boundary is by default enforced at runtime, but can be disabled through the `Runtime_Protection` property.

(3) Threads contained in a process execute within the virtual address space of the process.

(4) Processes may contain subprogram subcomponents. The code of such subprograms resides in the address space of the process. The calling semantics to such subprograms are defined in Section XX.

(5) A process may contain mode declarations. In this case, each mode can represent a different configuration of contained threads, their connections, and mode-specific property associations. The transition between modes is determined by the mode transition declarations and is triggered by the arrival of *mode transition trigger events* (see Sections 12 and 13.6).

(6) The associated source text for each process is compiled and linked to form binary images in accordance with the applicable source language standard. These binary images must be loaded into memory before any thread contained in a process can execute, i.e., enter its *perform thread initialization* state.

(7) The time to load binary images into the virtual address space of a process is bounded by the `Load_Deadline` and `Load_Time` properties. The failure to meet these timing requirements is considered an error.

(8) The process states, transitions, and actions are illustrated in Figure 8. Once a processor of an execution platform is started, binary images making up the virtual address space of processes bound to the processor are ready to be loaded, which is indicated by **started(processor)**. If the process is bound to a virtual processor of a processor, then process loading begins when the virtual processor is started, which is indicated by **started(vprocessor)**. Loading may take zero time for binary images that have been preloaded in ROM, PROM, or EPROM. Completion of loading, which is indicated by **loaded(process)**, triggers threads to be initialized (see Figure 5).

(9) A process, i.e., its contained threads, can be stopped (also known as a deferred abort), which is indicated by **stop(process)** , or by stopping the virtual processor or processor to which the process is bound. A process is considered stopped when all threads of the process are halted, are awaiting a dispatch, or are not part of the current mode and have executed their finalize entrypoint.

(10) A process, i.e., its contained threads, can be aborted, which is indicated by **abort(process)**. In this case, all contained threads terminate their execution immediately and release any resources (see Figure 5, Figure 6, and Figure 7).

© SAE International DRAFT

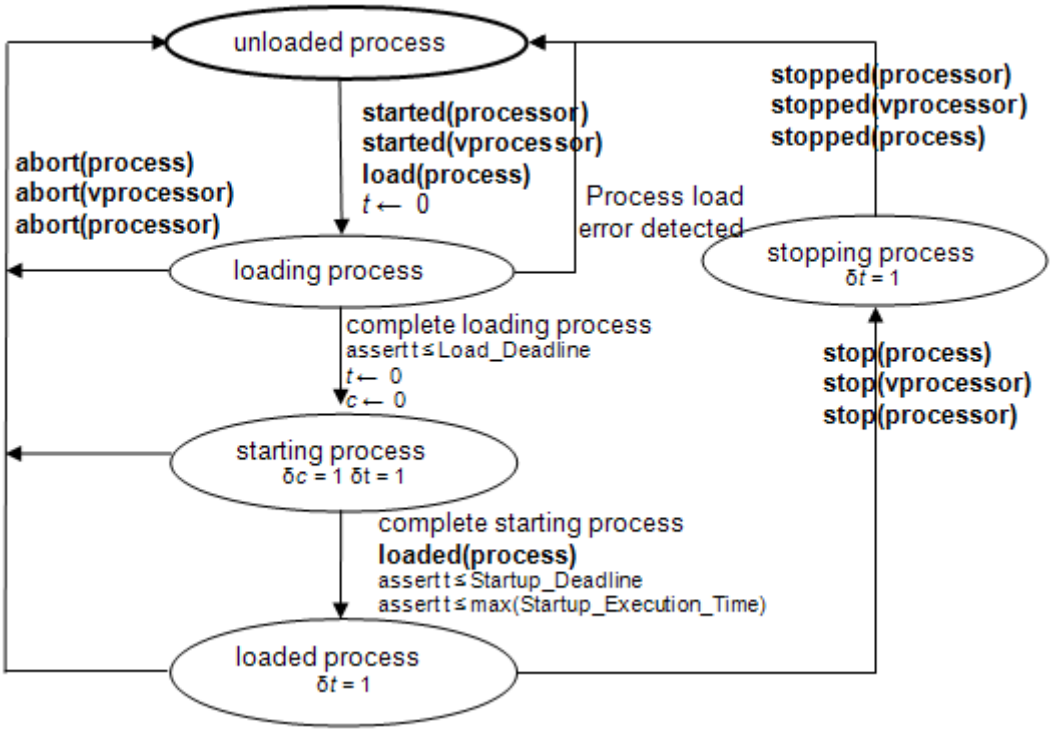

Figure 4 − Process States and Actions

Processing Requirements and Permissions

(11) A method of implementation must link all associated source text for the complete set of subcomponents of a process, including the process component itself and all actual subcomponents specified for required subcomponents. This set of source compilation units must form a single complete and legal program as defined by the applicable source language standard. Linking of this set of source compilation units is performed in accordance with the applicable source language standard for the process.

(12) If the applicable source language standard used to implement a component permits a mixture of source languages, then subcomponents may have different source language property values.

(13) This standard permits dynamic virtual memory management or dynamic library linking after process loading has completed and thread execution has started. However, a method for implementing a system must assure that all deadline properties will be satisfied to the required level of assurance for each thread.

NOTE: An AADL process represents only a virtual address space and requires at least one explicitly declared thread subcomponent in order to be executable. The explicitly declared thread in AADL allows for unambiguous specification of port connections to threads. In contrast, a POSIX process represents both a protected address space and an implicit thread.

3. EXECUTION PLATFORM COMPONENTS

(14) This section describes the categories of execution platform components that represent computing hardware: processor, virtual processor, memory, bus, virtual bus; and the physical environment: device.

(15) Processors can execute threads. Processors can contain memory subcomponents. Processors can access memories and communicate with devices and other processors over buses. Threads, thread groups, and processes are bound to processors.

© SAE International

(16) Virtual processors are logical execution platform elements that can execute threads. Threads, thread groups, and processes can be bound to virtual processors. Virtual processors must be bound to or contained in processors. This determines the binding of threads to processors.

(17) Memories represent randomly addressable storage capable of storing binary images in the form of data and code. Memories can be accessed by executing threads.

(18) Buses support physical communication between processors, devices, and memories. A bus provides the resources necessary to perform exchanges of control and data as specified by connections. These resources include bandwidth and protocols to perform the exchange. A connection may be bound to a sequence of buses and intermediate processors and devices in a manner that is analogous to the binding of threads to processors.

(19) Virtual buses represent a logical bus abstraction to model protocols and virtual channels. Virtual buses can be contained in or bound to processors and buses. Connections can specify that they require specific protocols, or certain quality of service from protocols of platform components they are bound to.

(20) Devices represent entities of the physical environment, e.g., an engine, or entities that interface with the physical environment, e.g., a sensor or an actuator. A device can interact with application software components through their port and provides subprogram access features. A device may interact with other execution platform components through bus access connections. A device may achieve its functionality through device internal software or may require device driver software to be executed by a processor. Binary images or threads cannot be bound to devices.

(21) Processors may include software that implements the capability of the processor to schedule and execute threads and other services of the processor. Its source text and data in the form of binary images will be bound to memories accessible from that processor. The resource requirements of this software are reflected in processor properties.

(22) Execution platform components can be assembled into execution platform systems, i.e., into systems of execution platform components to model complex physical computing hardware components and software/hardware computing systems, through the use of system components (see Section 7.1). The execution platform systems and their components may denote physical computing hardware for example, memory to represent a hard disk or RAM. Execution platform systems may also model abstracted storage, for example, a device or memory to represent a database, depending on the purpose of the model.

(23) The hardware represented by the execution platform components may be modeled in a hardware description or simulation language. Alternatively, it may be represented using configuration data for programmable logic devices. A simulation may be used to characterize the components. Such descriptions can be associated with the component by property association.

(24) Execution platform components can represent high-level abstractions of physical and computing components. A detailed AADL model of their implementation can be represented by system implementations that are associated with the execution platform component by property (see Section 14). This effectively models a layered system architecture.

3.1 Processors

(25) A processor is an abstraction of hardware and software that is responsible for scheduling and executing threads and virtual processors that are bound to it. A processor also may execute driver software that is declared as part of devices that can be accessed from that processor. Processors may contain memories and may access memories and devices via buses.

Semantics

(26) Processor states and transitions are illustrated in the hybrid automaton shown in Figure 9. The labels in this hybrid automaton interact with labels in the system hybrid automaton (see Figure 22), the virtual processor hybrid automaton (see Figure 10), and the process hybrid automaton (see Figure 8).

- The initial state of a processor is *stopped*.
- When a processor is started, it enters the *processor starting* state. In this state, the processor hardware is initialized and any processor software that provides thread scheduling functionality is loaded into memory and initialized. Note

© SAE International DRAFT

that the virtual address space load images of processes may already have been loaded as part of a single load image that includes the processor or virtual processor software.

- Once the processor and its software for handling virtual processors or processes is initialized, the processor transitions to the *processor operational* state with **started(processor)**. At this point virtual processor, process, and in turn thread initialization will start.

(27) As a result of a processor abort, any threads bound to the processor are aborted, as indicated by **abort(processor)** in the hybrid automaton in Figure 9 and in the hybrid automata figures in Sections 5.4 and 5.6. After that any virtual processor bound to or contained in a processor is aborted.

(28) A stop processor request results in a transition to the *processor stopping* state at the next hyperperiod of the periodic threads bound to the processor or to its virtual processors. The length of the hyperperiod can be reduced by using the `Synchronized_Components` property to minimize the number of periodic threads that must be synchronized within the hyperperiod (see Section 12).

(29) When the next hyperperiod begins, the virtual processors and processes with threads bound to the processor are informed about the stoppage request, as indicated by **stop(processor)** in the hybrid automaton in Figure 9. The process hybrid automaton (see Figure 8) in turn causes the thread hybrid automaton to respond, as indicated with **stop(processor)** in the hybrid automaton in Figure 5. In this case, any threads bound to the processor are permitted to complete their dispatch executions and perform any finalization before the process is stopped, which is indicated by **stopped(process)** in Figure 8.

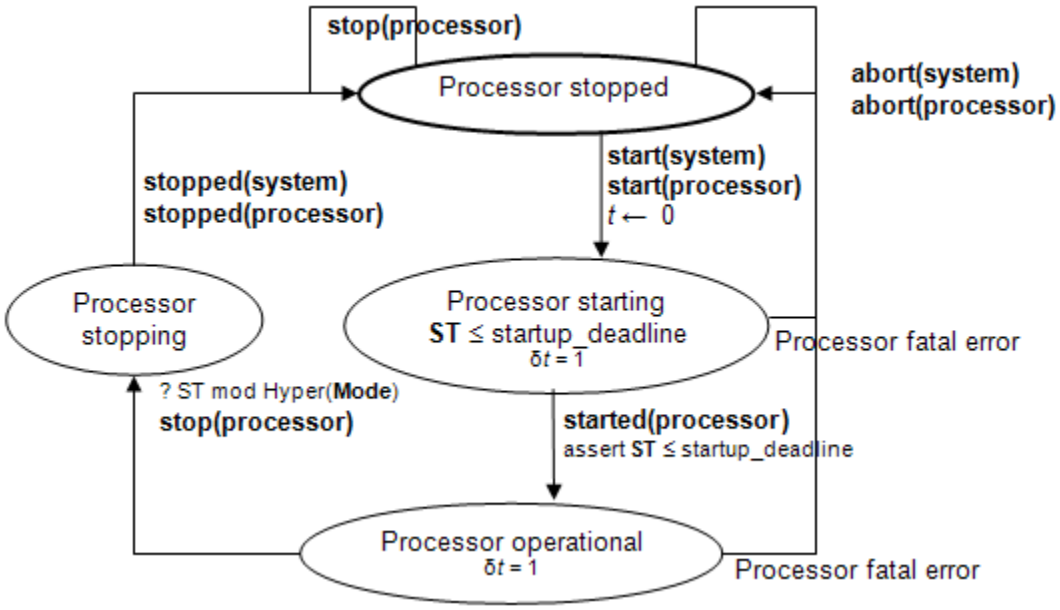

Figure 5 − Processor States and Actions

(30) A processor may have ports through which it reports information to the application software, e.g., to report error conditions. Those ports are identified in port connection declarations by the reserved word **processor** (see Section 9.1).

(31) A processor may have provide subprogram or subprogram group access to represent services that can be called by the application. A subprogram call identifies such a service by the subprogram classifier and the binding to the specific processor is implicit through the actual processor binding of the thread that contains the call.

(32) A processor component can include protocols in its abstraction. These protocols can be explicitly modeled as virtual bus subcomponents to satisfy protocol requirements by connections. The fact that a protocol is supported by a processor is specified by a `Provided_Virtual_Bus_Class` property.

© SAE International

(33) A processor can contain a bus subcomponent that it makes externally accessible. This models a bus that is managed by the processor and other components can connect to it. In this case the processor is implicitly connected to this bus.

(34) Different processors may be different execution speeds. This affects the execution time specified for threads and subprograms. The **in binding** statement of property associations permits processor type specific execution times to be declared. The execution time of a thread or subprogram can also be scaled according to scaling factors between different processors. The `Processor_Capacity property` specifies the speed of a processor for resource budget analysis. This property is also used to determine a scaling factor for execution time with respect to the processor capacity of a specified `Reference_Processor`.

(35) The processor is an abstraction of a hardware processor and possibly a runtime system. If it is desirable, the internals of the processor can be modeled by AADL as a system in its own right, i.e., an application system and an execution platform. This system implementation description can be associated with the processor component declaration by the `Implemented_As` property (see Section 14).

Processing Requirements and Permissions

(36) A method of implementation is not required to monitor the startup deadline and report an overflow as an error.

(37) A method of implementation may choose to enforce thread deadlines and maximum compute execution time. Violations are reported as thread recoverable errors.

3.2 Virtual Processors

(38) A virtual processor represents a logical resource that is capable of scheduling and executing threads and other virtual processors bound to them. Virtual processors can be declared as a subcomponent of a processor or another virtual processor, i.e., they are implicitly bound to the processor or virtual processor they are contained in. Virtual processors can also be declared separately, that is as a subcomponent of a system component, and explicitly bound to a processor or virtual processor. This allows virtual processors to represent hierarchical schedulers that schedule thread and/or virtual processors bound to them. In a fully bound system every virtual processor and thread is eventually bound to a physical processor. Virtual processors can be interconnected via virtual buses. This allows users to model virtual platforms.

Consistency Rules

(C1) In a fully deployed system a requires virtual bus binding of a virtual processor specified by the `Required_Virtual_Bus_Class` property must be satisfied by binding the virtual processor to a virtual processor or processor that provides this virtual bus. It is also satisfied if the virtual processor is contained in a processor and the respective virtual bus is bound to the processor.

Semantics

(39) Virtual processor states and transitions are illustrated in the hybrid automaton shown in Figure 10. The hybrid automaton of a virtual processor interacts with the hybrid automaton of the processor or virtual processor that it is bound to. A virtual processor is permitted to initialize itself after the processor and any virtual processors are initialized, and before any processes or threads are initialized. Similarly, virtual processors are stopped after threads, processes, and virtual processors contained in them are stopped.

© SAE International

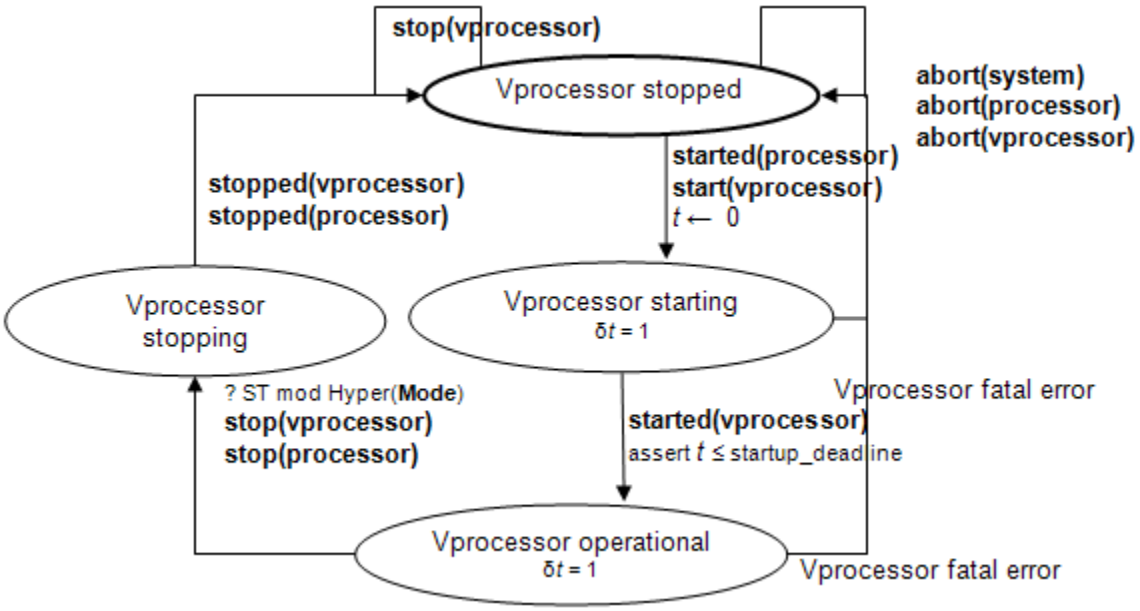

Figure 6 − Virtual Processor States and Actions

(40) The virtual processor is a logical abstraction of a processor. If it is desirable, the internals of the virtual processor can be modeled by AADL as an application system in its own right. For example, a virtual processor may represent an operating system that can be described in terms of processes and threads. This system implementation description can be associated with the device component declaration by the `Implemented_As` property (see Section 14).

3.3 Memory

(41) A memory component represents an execution platform component that stores code and data binaries. As such, it can be viewed as a passive component: it has no dynamic semantics attached by default. A design may attach system-specific semantics through fault or transactional modeling.

3.4 Buses

(42) A bus component represents an execution platform component that can exchange control and data between memories, processors, and devices. This execution platform component represents a communication channel, typically hardware together with communication protocols. Supported communication protocols can be explicitly modeled through virtual buses (see Section XX).

(43) The semantics of a bus is not prescribed, a bus is assumed to be a perfect lossless communication channel. More precise semantics may be defined project-specific properties, or by attaching behavioral and error models.

Processing Requirements and Permissions

(44) A method of implementation shall define how the final size of a transmission is determined for a specific connection. Implementation choices and restrictions such as packetization and header and trailer information are not defined in this standard.

(45) A method of implementation shall define the methods used for bus arbitration and scheduling. If desired this can be modeled by a notation of your choice and associated with a bus via property. Alternatively, it can be modeled through an AADL system model and associated with the bus through an `Implemented_As` property.

Examples

DRAFT © SAE International

3.5 Virtual Buses

(46) A virtual bus component represents logical bus abstraction such as a virtual channel or communication protocol.

(47) The semantics of a virtual bus is not prescribed, a virtual bus is assumed to be a perfect lossless communication channel. More precise semantics may be defined project-specific properties, or by attaching behavioral and error models.

Examples

3.6 Devices

(48) A device component represents a physical or mechanical component within the system, entities in the external environment. A device is logically connected to application software components and physically connected to computing hardware. Examples of devices are sensors, actuators, cameras, brakes.

(49) The semantics of a device is not prescribed. It is assumed to be specified depending on the modeling objective. For instance, definition and semantics of a device would be different for a code generation, an error modeling or a latency analysis perspective.

Examples

4. SYSTEM COMPOSITION

(50) Systems are organized into a hierarchy of components to reflect the structure of actual systems being modeled. This hierarchy is modeled by *system* declarations to represent a composition of components into composite components. A *system instance* models an instance of an application system and its binding to a system that contains execution platform components.

4.1 Systems

(51) A system component represents an assembly of software and execution platform components. All subcomponents of a system are considered to be contained in that system. As an enclosing entity, a system controls the lifecycle of its subcomponents.

Dynamic Semantics

(52) The semantics of a system comprises three ;major phases: startup, nominal activities and mode changes

4.1.1 System Startup

(53) On system startup the system instance transitions from *system offline* to *system starting* (see Figure 22). **Start(system)** initiates the initialization of the execution platform components. In the case of processors, the binary images of the kernel address space are loaded into memory of each processor, and execution is started to initialize the execution platform software (see Figure 9). Loading into memory may take zero time, if the memory can be preloaded, e.g., PROM or flash memory.

(54) Once a processor is initialized (*Processor operational*), each processor initiates the initialization of virtual processors bound to the processor, if any (see Figure 9). When the virtual processors have completed their initialization they are operational (see Figure 10).

(55) When processors and virtual processors have entered their operational state (**started(processor)** and **started(vprocessor)**), the loading of the binary images of processes bound to the specific processor or its virtual processors into memory is initiated (see Figure 8). Process binary images are loaded in the memory component to which the process and its contained software components are bound. In a static process loading scenario, all binary images must be loaded before execution of the application system starts, i.e., thread initialization is initiated. In a

DRAFT © SAE International

dynamic process loading scenario, binary images of all the processes that contain a thread that is part of the current mode must be loaded

(56) The maximum system initialization time can be determined as

- max(`Startup_Deadline`) of all processors and other hardware components
- + max(`Startup_Deadline`) of all virtual processors
- + max(`Load_Deadline`) of all processes
- + max(Process_Startup_Deadline) of all processes
- + max(`Initialize_Deadline`) of all threads.

(57) All software components for a process must be bound to memory components that are all accessible from every processor to which any thread contained in the process is bound. That is, every thread is able to access every memory component into which the binary image of the process containing that thread is loaded.

(58) Data components shared across processes must be bound to memory accessible by all processors to which the processes sharing the data component are bound.

(59) Thread initialization must be completed by the next hyperperiod of the initial mode. Once all threads are initialized, threads that are part of the initial mode enter the await dispatch state. If loaded, threads that are not part of the initial mode enter the suspend awaiting mode state (see Figure 5). At their first dispatch, the initial values of connected **out** or **in out** ports are made available to destination threads in their **in** or **in out** ports.

(60) This initialization model assumes independent initialization of threads, i.e., no ordering requirement (other than processes must be initialized first). If there is an ordering requirement the user can introduce an initialization mode, in which they can utilize the full power of AADL to specify thread execution dependencies.

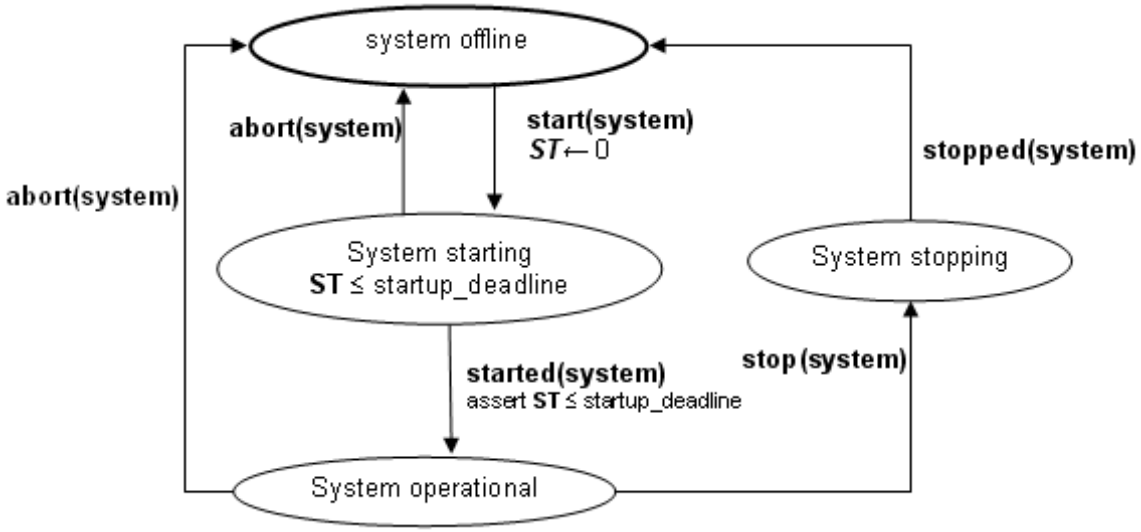

Figure 7 − System Instance States, Transitions, and Actions

4.1.2 Normal System Operation

(61) Normal operation, i.e., the execution semantics of individual threads and transfer of data and control according to connection and shared access semantics, have been covered in previous sections. In this section we focus on the coordination of such execution semantics throughout a system instance.

(62) A system instance is called synchronized if all components use a globally synchronized reference time. A system instance is called asynchronous if different components use separate clocks with the potential for clock drift. The clock drift in asynchronous systems may be bounded, e.g., by resynchronizing the clocks.

© SAE International

(63) In a synchronized system, periodic threads are dispatched simultaneously with respect to a global clock. The hyperperiod of a set of periodic threads is defined to be the least common multiple of the periods of those threads.

(64) In a synchronized system, a raised event logically arrives simultaneously at the ultimate destination of all semantic connections whose ultimate source raised the event. In a synchronized system, two events are considered to be raised logically simultaneously if they occur within the granularity of the globally synchronized reference time. If several events are logically raised simultaneously and arrive at the same port or at different transitions out of the current mode in the same or different components, the order of arrival is implementation-dependent.

(65) In an asynchronous system the above hold within a synchronization domain, i.e., periodic threads are dispatched simultaneously and events arrive logically simultaneously. A method of implementing a system may provide coordination protocols for asynchronous system to provide simultaneity guarantees within a certain time granularity. Otherwise, the dispatches and event arrivals are considered independent across synchronization domains.

4.1.3 System Operation Modes

(66) The set of all modes specified for all components of a system instance form a set of concurrent mode state machines, whose composite state is called *system operation mode* (SOM). The set of possible SOMs is the cross product of the sets of modes for each component. That is, a SOM is a set of component modes, one mode for each component of the system. The initial SOM is the set of initial modes for each component.

(67) The set of possible SOMs can be reduced by taking into account mode combinations that are not reachable. For example, if a component instance with modes is itself not active under certain modes of one of its containing components, then these mode combinations can be excluded. Similarly, of a component inherits modes from a containing component, then only the mode combinations of the containing component need to be considered. Finally, reachability analysis of modes taking into account mode transition conditions may further reduce the set of SOMs to be considered.

(68) The discrete variable **Mode** denotes a SOM. That is, the variable **Mode** denotes a compound discrete state that is defined by the mode hybrid semantic diagrams for each component instance in the system. Note that the value of **Mode** will in general change at various instants of time during system operation, although not in a continuous time-varying way.

4.1.4 System Operation Mode Transitions

(69) A SOM transition is initiated whenever a mode transition trigger event (see also Section 12) occurs in any modal component in the system instance. A single sent event or event data can trigger a mode switch request in one or more components. In a synchronized system, this trigger event occurs logically simultaneously for all components and the resulting component mode transition requests are treated as a single SOM transition.

(70) Several events may occur logically simultaneously and may originate from different ports of the same component, e.g., through a `Send_Output` service call with multiple ports as parameter, or from different components at the same time. If they are semantically connected to transitions in different components that lead out of their current mode or to different transitions out of the same mode in one component, then events are treated as independent SOM transition initiations.

(71) If multiple SOM transition initiations occurs logically simultaneously, one is chosen in an implementation-dependent manner. If an `Urgency` property is associated with each port named in mode transitions, then the mode transition with the highest port urgency takes precedence. If several ports have the same urgency then the mode transition is chosen non-deterministically.

(72) A mode transition of a thread internal mode, i.e., a mode declared in the thread or one of its subprograms, that is triggered by the component itself or is triggered by an event or data arriving ata port of the thread, takes place at the next thread dispatch. For further detail see Section 5.4.5.

(73) A mode transition of a processor, device, bus, or memory internal mode, i.e., a mode declared in the component implementation, that is triggered by the component itself or is triggered by an external event, takes place immediately.

The next paragraphs address mode transitions that involve activation and deactivation of threads and connections.

DRAFT © SAE International

(74) If a mode transition has a `Mode_Transition_Response` property value of `Emergency`, the `mode_transition_in_progress` state is entered in zero time and the actual SOM transition is performed immediately. In this case all threads in the performing state that have to be deactivated are aborted.

(75) After an event triggering a SOM transition request has arrived, the actual SOM transition occurs as a `Planned` response, if the mode transition does not have a `Mode_Transition_Response` property value of `Emergency`. In this case, execution continues under the old SOM until the dispatches of a critical set of periodic components (threads and devices), which are active in the old SOM, align at their hyperperiod. At that point the `mode_transition_in_progress` state is entered. A component is considered critical if it has a `Synchronized_Component` property value of true. If the set of critical components is empty, the `mode_transition_in_progress` state is entered immediately. This is indicated in Figure 23.

(76) by the guard of the function Hyper(**Mode**) on the transition from the `current_system_operation_mode` state to the `mode_transition_in_progress` state.

(77) Until the `mode_transition_in_progress` state is entered, any mode transition trigger event with the same or lower urgency that would result in a SOM transition from the current SOM is ignored. A trigger event with higher urgency supersedes the event under consideration for determining the SOM transition at the time the `mode_transition_in_progress` state is entered.

(78) A runtime transition between SOMs requires a non-zero interval of time, during which the system is said to be in transition between two system modes of operation. While a system is in transition, excluding the instants of time at the start and end of a transition, all mode transition trigger events are ignored with respect to mode transitions.

(79) The system is in a *mode transition in progress state* for a limited amount of time. This is determined by the maximum time required to perform all deactivations and activations of threads and connections (see below), increased to the next multiple of the hyperperiod of a non-zero set of critical components that continue to execute, i.e., periodic threads that are active in the old and in the new SOM and have a `Synchronized_Component` property value of true. This is shown in Figure 23.

(80) by the guard of the function Hyper(**Mode**)* on the transition from the `mode_transition_in_progress` state to the `current_system_operation_mode` state. After that period of time, the component is considered to operate in the new mode and the active threads in the new mode start to execute.

(81) At the time the `mode_transition_in_progress` state is entered there are several kinds of threads

- *Continuing threads*: threads that continue to execute because they are active in the old mode and active in the new mode;
- *Activated threads*: threads that are inactive in the old mode and are active in the new mode;
- *Deactivated threads*: threads that are active in the old mode, not active in the new mode, and whose dispatches align at the hyperperiod;
- *Zombie threads*: aperiodic, sporadic, timed, or hybrid threads as well as periodic threads that are active in the old mode and not active in the new mode and whose dispatches are not aligned at the hyperperiod, i.e., they may still be in the perform thread computation state (see Figure 5) and may have events or event data in their port queues.

(82) At the instant of time the `mode_transition_in_progress` state is entered, connections that are part of the old SOM and not part of the new SOM are disabled.

(83) While in the `mode_transition_in_progress` state, for all connections between data ports that are declared to be active in the current mode transition and whose source threads have completed execution at the time the `mode_transition_in_progress` state is entered, data is transferred from the **out** data port such that its value becomes available at the first dispatch of the receiving thread.

(84) At the instant in time the `mode_transition_in_progress` state is exited, connections that are not part of the old SOM and are part of the new SOM are enabled. At that time the new current SOM is entered and the following hold for data port connections. The data value of the **out** data port of the source thread is transferred into the **in** data port

DRAFT © SAE International

variable of the newly enabled thread, unless the **in** data port of the destination thread is the destination of a connection active in the current mode transition.

(85) While in the `mode_transition_in_progress` state, all continuing threads continue to get dispatched and execute, but will only use connections that are active in both the old and the new SOM.

(86) When the `mode_transition_in_progress` state is entered, *thread* **exit(Mode)** is triggered to deactivate all threads that are part of the old mode and not part of the new mode, i.e., the set of deactivated threads. This results in the execution of deactivation entrypoints for those threads (see Figure 5).

(87) When the `mode_transition_in_progress` state is entered, *thread* **enter(Mode)** is triggered to activate threads that are part of the new mode and not part of the old mode, i.e., the set of activated threads. This permits those threads to execute their activation entrypoints (see Figure 5).

(88) There is no requirement that all deactivation entrypoint executions complete before any activation entrypoint executions start. The maximum execution time for the deactivations and activations is the maximum deadline of the respective entrypoints.

(89) Zombie threads may be stopped at actual mode switch time, they may be suspended, or they may complete their execution while mode transition is in progress. The `Active_Thread_Handling_Protocol` property specifies for each such thread what action is to be taken at the time the `mode_transition_in_progress` state is entered.

(90) The default action is to `stop` the execution of the zombie thread and execute its recover entrypoint indicating an stop for deactivation - effectively handling them like deactivated threads. This permits the thread to recover to a consistent state for future activation, including the release of resources held by the thread. Upon completion of the recover entrypoint, execution the thread enters the suspended awaiting mode state; event and event data port queues of the thread are flushed by default or remain in the queue until the thread is activated again as specified by the `Active_Thread_Queue_Handling_Protocol` property. If the thread was executing a remotely called subprogram, the current dispatch execution of the calling thread of a call in progress or queued call is also aborted. In this case the recover entrypoint deadline is taken into account when determining the duration of the `mode_transition_in_progress` state.

(91) Other actions are project-specific implementor provided action and may include:

- `Suspend` the execution of the zombie thread for resumption the next time the thread is part of a new mode. This action is only safe to use if the thread does not hold the lock to a shared resource.
- `Complete_one` permits the thread to complete the execution of its current dispatch and invoke its deactivation entrypoint. Any remaining queued events, or event data may be flushed, or remain in the queue until the thread is activated again, as specified by the `Active_Thread_Queue_Handling_Protocol` property. The output is held and communicated according to the connections when the thread is reactivated. In this case the execution of the zombie thread competes for execution platform resources.
- `Complete_all` permits the thread to finish processing all events or event data in its queues at the time the actual mode switch state is entered. The output is held and communicated according to the connections when the thread is reactivated. When execution completes the deactivation entrypoint is executed. In this case the execution of the zombie thread competes for execution platform resources.

(92) At the next multiple of the SOM transition hyperperiod the system instance enters `current_system_operation_mode` state and starts responding to new requests for SOM transition.

DRAFT © SAE International

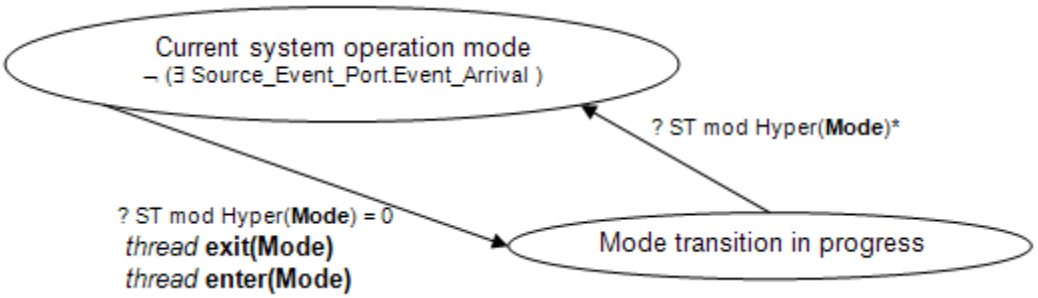

Figure 8 − System Mode Transition Semantics

(93) The synchronization scope for **enter(Mode)** consists of all threads that are contained in the system instance that were inactive and are about to become active. The synchronization scope for **exit(Mode)** contains all threads that are contained in the system instance that were active and are to become inactive. The edge labels **enter(Mode)** and **exit(Mode)** also appear in the set of concurrent semantic automata derived from the mode declarations in a specification. That is, **enter(Mode)** and **exit(Mode)** transitions for threads occur synchronously with a transition from the `current_system_operation_mode` state to the `mode_transition_in_progress` state.

The description of this time-coordinated transitioning of system operation mode assumes a synchronous system, i.e., a single synchronization domain. In the case of an asynchronous system, multiple synchronization domains exist in a system. In this case, the coordinated activation and deactivation of threads and connections as part of a system operation mode transition must be ensured within each synchronization domain. In the case of an asynchronous system, coordination protocols may be supported to coordinate system operation mode transition across synchronization domains within bounded time

DRAFT © SAE International