Modes in Asynchronous Systems

Jean-François Rolland, Jean-Paul Bodeveix, M Filali, David Chemouil, Dave Thomas

▶ **To cite this version:**

Jean-François Rolland, Jean-Paul Bodeveix, M Filali, David Chemouil, Dave Thomas. Modes in Asynchronous Systems. 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008), Mar 2008, Belfast, Ireland. pp.282 - 287, ⟨10.1109/ICECCS.2008.28⟩. ⟨hal-00784986⟩

HAL Id: hal-00784986

https://hal.science/hal-00784986v1

Submitted on 15 Dec 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Modes in asynchronous systems

Rolland Jean-François [*]
IRIT, Université Paul Sabatier
Toulouse
rolland@irit.fr

Bodeveix Jean-Paul
IRIT, Université Paul Sabatier
Toulouse
bodeveix@irit.fr

Filali Mamoun
IRIT, Université Paul Sabatier
Toulouse
filali@irit.fr

Chemouil David
CNES
Toulouse

Thomas Dave
ASTRIUM EADS
dave.thomas@astrium.eads.net

Abstract

In this paper we study the mode concept in asynchronous systems. First, we propose an abstract TLA+ specification. Then, we discuss how the mode concepts proposed by the two architecture languages: Giotto and AADL could be related to this abstraction.

1 Introduction

The use of models is now acknowledged as a first step for the production of correct code. For such a purpose, the TOPCASED project [14], provides generic solutions to elaborate domain specific models. For real time and embedded systems, the AADL[1] [2] architecture description language is currently supported [14]; actually AADL textual and graphical editors are provided; moreover bridges to different analysis tools are also available within the toolkit. Besides model driven engineering, one of the aims of the TOPCASED project is to promote the production of certified code, i.e., code with a proof of its correctness. In the avionics domain, the synchronous approach [3] has already provided solutions towards this end: actually, certified code generators do exist and are already in use [5]. However, with respect to the asynchronous approach, to the best of our knowledge, no acknowledged solution does exist. From our point of view, two reasons can explain such a situation:

- Although, the asynchronous approach is widely used in practice, we lack abstractions that will be useful for analysis purposes. Indeed, the asynchronous approach is too nondeterministic to lend itself to usual model checking techniques.
- The basic mechanisms are complex. It follows that it is not easy to provide a coherent execution model that will handle the needs of real time applications.

With respect to these two points, one the major outcomes of AADL [12] has been to capitalize more than 10 years of experiments based on MetaH[15] and to propose an execution model that covers most of the needs of real-time systems while preserving analysis perspectives. Since one of the goals of the TOPCASED project is to promote the production of certified code, all the mechanisms underlying the execution model must be semantically well defined. In a previous paper [4], we have been interested in the communication aspects. In this paper, we address the mode mechanisms provided by AADL.

Real time systems and especially embedded ones are well known to be static, e.g. threads are not created dynamically, memory is statically allocated, ... The notion of mode has been proposed to express the fact that although the configuration of a real time system should be known a priori, it can change in a statically defined way. In this paper, we propose an abstract specification of the mode concept in asynchronous systems. We try to assess this abstraction by considering how modes are dealt with in Giotto [7] and AADL.

In section 2 we present an abstract specification of modes. Then, in section 3, we present two concrete

[*] Work funded by CNES and EADS Astrium Satellites

[1] AADL stands for Architecture Analysis and Design Language.

models related to this abstract specification. In section 4, we conclude and present some perspectives of our work.

2 An abstract specification of modes

Although mode's description in the AADL standard is quite precise, it seems opportune for us to formally specify their behavior. This work has two main goals :

- describe precisely the behavior of a system during a mode transition,
- and possibly propose evolutions of the language.

We develop several specifications incrementally, we present here only the most complete one. Aiming to keep models simple, we try to focus only on the mode behavior, thus we abstract communication and scheduling. In this model we will take into account only one type of messages, those that are involved in mode switching. As we do not describe the behavior of threads or any components we will consider that those messages can appear in the system. The scheduling mechanism is reduced to a two state automaton. A thread can be either active or idle, the transition between those two states is atomic and non deterministic. With those specifications we try to list the different mechanisms that can be used to describe a mode transition. Those mechanisms are used to define precisely the time of the mode switch and the way threads are handled.

2.1 Atomic mode transitions

At this level, we introduce modes, the events that fire transitions between modes and for each mode the threads that are allowed to run. The mode automaton is encoded by constants, modes states are encoded by a set, transitions are defined by a relation between the couple `<old mode, event>` and the new mode. The configuration of a mode is defined by a relation that associates a set of threads to each mode. This level is characterized by the invariant stating that the current set of active threads, defined as the variable `currentThreads`, is always a subset of the threads allowed by the mode:

$$\text{currentThreads} \in \mathbf{SUBSET}\ \text{ModeThreads}[\text{currentMode}]$$

A mode transition is triggered by the reception of an event. We consider that a mode transition is atomic and is executed as soon as the event is received. During the transition all the threads of the old mode not present in the new mode are stopped and the ones of the new mode are started.

2.2 Breaking transitions

In fact, considering transitions as atomic is not realistic. Indeed, mode transitions are complex and do take time. Consequently, we "break" the previous mode transitions. The main interest of this level, is to take into account that we do not switch instantaneously from the thread set of the current mode to the thread set of the next mode. We have an intermediate time during which the threads that are no longer in the thread set of the next mode "disappear" gradually. The intermediate time ends when only threads belonging to the next mode remain.

2.3 Critical threads

The aim of this level is to make precise that some threads should not disappear when a mode transition occurs. More precisely, the mode switch is delayed until critical threads that do not belong to the next mode complete. At this level we distinguish two types of threads, normal threads and critical threads. On the mode transition normal threads that are not part of the new mode are stopped if they are active. Critical threads must be stopped when they become idle. When this event is received the first thing the system does is to keep running normally. In this first phase, the system waits for the end of all critical threads. When all critical threads are idle the mode transition is executed and the system comes back to normal mode.

2.4 Zombie threads

When an event mode occurs, some threads have to disappear since they do not belong to the thread set of the next mode. However, before disappearing some cleaning staff may be necessary, e.g., release some resources they currently hold. For such a purpose, we introduce the so called zombie threads. The introduction of this type of threads constrains us to add a third state in our state machine. In this state the system waits for the termination of zombie threads. The rest of the system behaves normally in the new mode. This state will be left when all zombie threads will have finished their execution. The utilization of such threads implies an overhead on the processor load. Consequently, they must be used carefully.

2.5 Preemption and priorities

At this level we add two mechanisms to our specification. In the first one we slightly modify the definition of critical threads. In the precedent specifications for each mode we defined a set of critical threads. But this set may also depend on the type of mode transition. For example a mode switch due to an hardware error must be handled faster than a planned mode switch. Thus, the set of critical threads depends on the urgency of the mode switch. This is why the set of critical thread does not depend on modes only but also on mode transitions, i.e. the current mode and the event that triggered the transition.

This last feature consists in the introduction of a priority depending on the nature of the mode switch. We associate to each event a level of priority. A mode transition can be interrupted only if an event of higher priority arrives. If the system is waiting for the end of critical threads, this may change the set of critical threads. If the system is waiting for the end of zombie threads, all the zombie threads are stopped and the system starts to wait for the end of critical threads.

2.6 The TLA+ specification

We present here the most complete TLA+ [8] specification (see table 1). Figure 1 is a graphical representation of this automaton. The `Normal behavior` state is the initial state. The `ThreadTransition` is an abstraction of the scheduler: when this transition is executed some threads are activated (i.e. they enter in the `currentThreads` set), and other are deactivated (i.e. they leave the `currentThreads` set). This transition has a specific behavior when the system contains zombies. In this case a zombie can only leave the set of executing threads. The "start mode" transition mainly stores the event that have triggered the mode switch. The system remains in the old mode until `currentThreads` does not contain any critical threads. At this time the actual mode switch occurs. The `zombies` set is initialized, thread that are not allowed to be zombies in the next mode and that are not part of the new mode are stopped, and new threads are started. The `EndModeTransition` specifies that the system comes back in a normal mode only if all zombies have terminated their execution.

```
---------------------------- MODULE advanced_modes ----------------------------
EXTENDS Naturals

CONSTANTS
 Mode, InitialMode, Event, Thread, NextMode,
 domNextMode,  ModeThreads, Critical,
 Zombies, Priority

AllEvent ≜ Event ∪ {"NoEvent"}

ASSUME
  ∧ InitialMode ∈ Mode
  ∧ domNextMode ⊆ Mode × Event
  ∧ NextMode ∈ [domNextMode → Mode]
  ∧ ModeThreads ∈ [Mode → SUBSET Thread]
  ∧ "NoEvent" ∉ Event
  ∧ Critical ∈ [domNextMode → SUBSET Thread]
  ∧ Zombies ∈ [Mode → SUBSET Thread]
  ∧ ∀ m ∈ Mode : Zombies[m] ∩ ModeThreads[m] = {}
  ∧ Priority ∈ [AllEvent → Nat]
  ∧ Priority["NoEvent"] = 0 ∧ ∀ e ∈ Event : Priority[e] > 0

VARIABLES
 currentMode, currentThreads, currentEvent, zombies

TypeInvariant ≜
  ∧ currentMode ∈ Mode ∧ currentEvent ∈ AllEvent
  ∧ currentThreads ∈ SUBSET Thread
Invariant ≜
  ∧ currentThreads ∈
     SUBSET (ModeThreads[currentMode] ∪ zombies)
  ∧ zombies ∈ SUBSET Zombies[currentMode]

Init ≜
  ∧ currentMode = InitialMode
  ∧ currentThreads ∈ SUBSET ModeThreads[currentMode]
  ∧ zombies = {}
  ∧ currentEvent = "NoEvent"

StartModeTransition(evt) ≜
  ∧ ⟨currentMode, evt⟩ ∈ domNextMode
  ∧ Priority[evt] > Priority[currentEvent]
  ∧ currentEvent' = evt
  ∧ zombies' = {}
  ∧ currentThreads' = currentThreads \ zombies
  ∧ UNCHANGED ⟨currentMode⟩

ModeTransition ≜
  ∧ currentEvent ≠ "NoEvent"
  ∧ currentEvent' = "NoEvent"
  ∧ currentMode' = NextMode[currentMode, currentEvent]
  ∧ currentThreads ∩ Critical[currentMode, currentEvent] = {}
  ∧ zombies' = currentThreads ∩ Zombies[currentMode']
  ∧ currentThreads' ∈
 SUBSET (ModeThreads[currentMode'] ∪ zombies')

EndModeTransition ≜
  ∧ zombies = {}
  ∧ UNCHANGED ⟨currentMode, currentThreads, zombies, currentEvent⟩

ThreadTransition ≜
  ∧ currentThreads' ∈ SUBSET ModeThreads[currentMode]
  ∧ IF zombies ≠ {} THEN
     ∧ currentThreads' ∈
      SUBSET (ModeThreads[currentMode] ∪ zombies')
     ∧ zombies' ∈ SUBSET zombies
     ELSE
     ∧ currentThreads' ∈ SUBSET ModeThreads[currentMode]
     ∧ UNCHANGED ⟨zombies⟩
  ∧ UNCHANGED ⟨currentMode, currentEvent⟩
--------------------------------------------------------------------------------
```

Table 1. TLA modes specification

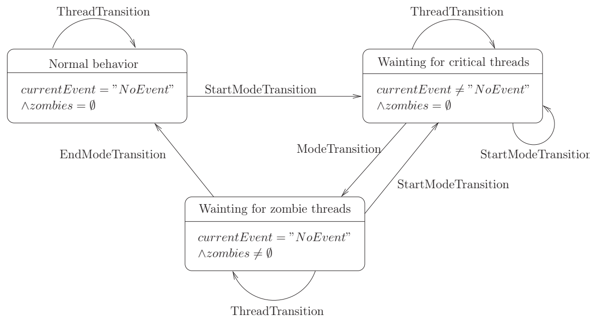

Figure 1. Abstract mode transitions

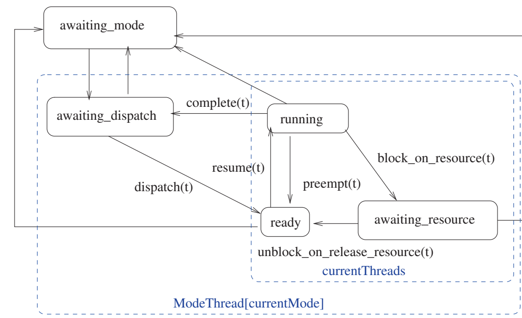

Figure 2. Abstraction of the thread behavior

3 Concrete models

In this section we try to define relations between our specification of the mode transition and languages that implement this function. The first comparison is made with Giotto, which can be seen as a synchronous approach of architecture languages. As its execution model is very strict the link between those two formalisms is quite easy to find. Secondly we will compare our work to the AADL vision of modes. As AADL was the starting point of this study the TLA specification matches quite well with the AADL execution model.

3.1 The Giotto refinement

Giotto is a time driven language, a system is a set of mode, each mode has a period and contains a set of threads. The period of a thread is defined by the number of activations of this thread in its mode and the period of the mode. At every period a condition is evaluated to decide if the system stays in the same mode or must change. The period of the mode corresponds to the hyper-period of the threads in the mode.

In our model the condition triggering mode switch is simply abstracted as the reception of an event. We don't evaluate a complex condition on data, we just test if an event has been received or not. The abstraction of the time at which the mode switch occurs, i.e. the period of the mode, is a little bit less simple. However, we can notice that the mode switch always occurs when all the threads have completed their execution (at their hyper-period). In our model this corresponds to a system where all the threads belong to the `CriticalThreads` set for all modes. In this case, the hyper-period corresponds to an empty `currentThread` set (all threads have finished their execution). Those constraints can be represented by the following conditions:

$$\forall\ m \in \text{Mode} : \text{Zombies}[m] = \emptyset$$
$$\forall\ m \in \text{Mode} : \forall\ e \text{ in Event} : \langle\langle m, e\rangle\rangle \in \text{domNextMode} \Rightarrow \text{CriticalThreads}[m, e] = \text{ModeThreads}[m]$$

3.2 The AADL refinement

In this part we first compare our work to the AADL notion of modes. Then we make two propositions of evolution for the standard. At last we bring up the problem of communication during mode transition.

3.2.1 AADL modes

The description of the mode automaton in AADL is distributed in the different hierarchical components, each component can contain a local mode automaton. The product of these automata is the mode automaton of the whole system. The composition of these automata is not studied in our model. We model the complete automaton as a set of constants and a variable. `Mode` is the set of all modes, `NextMode` is the set of all possible transitions, `Event` is the set of events that trigger mode switches, and `currentMode` is the variable called SOM (System Operational Mode) in the AADL standard. As we mainly focus on the behavior of the system during the mode switch we do not represent the behavior of threads. A thread can be waiting for a mode switch (the thread is not in the current mode), executing (the thread is in `currentThreads` state), or idle (the thread is in the current mode but not in `currentThread`).

In AADL like in Giotto the time of the mode switch depends on periodic threads. The main difference with the Giotto model is that not all periodic threads are used to determinate the hyper-period. A property

called `synchronized` is used to determinate if the thread must be taken into account to calculate the hyper-period. The mode switch occurs at the hyper-period of the synchronized threads. As we do not have the notion of time we will use the same abstraction as in Giotto, we can consider that at the time of the hyper-period all synchronized threads are waiting for their dispatch. The synchronized threads can be represented by our `CriticalThreads`. If there is no periodic threads the mode switch is immediate, the guard of the `ModeTransition` is true when an event arrives.

In AADL some threads of the old mode are allowed to end their execution in the new mode. It depends on the value of a specific property, `Active_thread_handling_protocol`. Threads of the old mode can be allowed to end their execution or to finish the computation of all the data contained in their ports. In the second case the thread will be dispatched numerous times. In our model we can allow a thread to end its execution in the new mode by adding it in the `Zombie` set of the new mode.

In AADL the system comes back in a nominal behavior one hyper-period after the mode switch. Before this deadline the system cannot start another mode switch. In our model we consider that the mode transition ends when all zombie threads have finished their execution.

3.2.2 Proposition of evolutions

In the current AADL standard we cannot handle priorities on modes transitions, i.e. a mode transition cannot be interrupted. Furthermore the set of synchronized threads depends only on the current mode. But those notions can be easily integrated using or modifying existing properties. The `synchronized` property associates a boolean to a thread in a particular mode. Thus the set of synchronized threads for a mode is fixed. In order to manage this set more precisely we can associate a set of mode's event ports to each thread. The `synchronized` property would define the set of transitions in which the thread would be considered as critical.

In order to define the priority of mode transitions we can simply use the `urgency` property on event ports.

3.2.3 Connections and mode switch

In this paper we have mainly focused on the behavior of a global system during a mode switch. But some other points are also very important. We need to describe precisely how threads communicate when the system is in

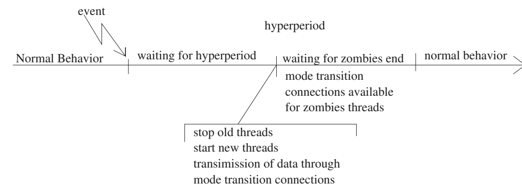

Figure 3. Time line of the mode transition

a mode transition. At the time of the actual mode switch connections of threads of the old mode are disabled and connections of the new mode are enabled. But managing communications on this way can generate orphan messages. We want to specify that the results of the last execution of threads of the old mode are taken into account in the new mode. Similarly we want to describe how are initialized ports of threads of the new mode. For those features AADL describes some specific connections called mode transition connections. The source of such a connection is a port of a thread of the old mode and the destination is a thread of the new mode. Those conditions are enabled at the time of the actual mode switch.

The second problem is related to threads of the old mode not present in the new one but still active (zombie threads). Connections of such a thread are deactivated but the thread is still computing results. The AADL standard does not precisely define how such a thread can communicate its result. The easiest way to specify a connection between those threads and threads of the new mode is to use mode transition connections.

3.2.4 Discussion

It is interesting to remark that mode mechanisms in asynchronous systems requires more attention than in synchronous systems [9, 13]; actually, since we do not assume the basic hypothesis of the synchronous approach: zero time computation, deterministic concurrency and instantaneous communication, we have to handle the transitional aspects related to these concepts. From our point of view, the formal specification of these aspects is challenging and is worth considering. In this article we do not take into account issues linked to scheduling and to shared ressources. Thoses points are studied in the following articles [11] [10] in an ADA framework.

4 Conclusion

In this paper, we have discussed about the mode concept in asynchronous systems. We have presented an abstract specification and assessed it through the mode notions available in AADL and Giotto. This specification can be seen as a list of the different possible behaviors of a system during mode transition. We have also presented the whole formal model of the abstraction which has been encoded in TLA.

We intend now to propose a formal specification of AADL modes as a refinement [1] of our abstraction. Another point that we intend to consider it the quantification of timing aspects with respect to mode changes. Actually, it appears that most timing properties are currently studied within a given mode; however little is said when considering mode transitions.

The presented models are too abstract to perform precise verifications with respect to timing. Going further requires to describe more precisely the scheduling of threads, and the communication model. We have already developed a model of the AADL execution platform that includes scheduling, communications and behaviors [6]. By integrating a refinement of the presented specifications in this model we will be able to perform timing evaluations of AADL models taking into account mode transitions. Thanks to these enhanced models, we expect to be able to make schedulability analysis of the system and to quantify the maximum duration of mode transitions.

References

[1] J.-R. Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.

[2] S. Aerospace. *SAE AS5506 : ARCHITECTURE ANALYSIS and DESIGN LANGUAGE (AADL).* SAE International, 2004.

[3] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[4] J.-P. Bodeveix, R. Cavallero, D. Chemouil, M. Filali, and J.-F. Rolland. A mapping from AADL to Java-RTSJ. In *International Workshop on Java Technologies for Realtime and Embedded Systems (JTRES), Vienna, Austria, 26/09/07-28/09/07*, ACM International Conference Proceeding Series, pages 165–174, http://www.acm.org/, septembre 2007. ACM.

[5] P. Caspi, C. Mazuet, R. Salem, and D. Weber. Formal design of distributed control systems with lustre. In *SAFECOMP '99: Proceedings of the 18th International Conference on Computer Computer Safety, Reliability and Security*, pages 396–409, London, UK, 1999. Springer-Verlag.

[6] R. B. França, J.-P. Bodeveix, M. Filali, J.-F. Rolland, D. Chemouil, and D. Thomas. The AADL behaviour annex – experiments and roadmap. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007), Auckland, New Zealand, 11/07/07-14/07/07*, pages 377–382, http://www.computer.org, 2007. IEEE Computer Society.

[7] T. Henzinger, C. Kirsch, M. Sanvido, and W. Pree. From control models to real-time code using Giotto. *IEEE Control Systems Magazine*, 23(1):50–64, January 2003.

[8] L. Lamport. *Specifying Systems:The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[9] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program.*, 46(3):219–254, 2003.

[10] J. Real, A. Burns, F. J. M. González, E. Schonberg, and A. Crespo. Dynamic ceiling priorities: A proposal for ada0y. In A. Llamosí and A. Strohmeier, editors, *Ada-Europe*, volume 3063 of *Lecture Notes in Computer Science*, pages 261–272. Springer, 2004.

[11] J. Real and A. J. Wellings. The ceiling protocol in multimoded real-time systems. In *Ada-Europe*, pages 275–286, 1999.

[12] G. Sébastien, P. Feiler, J.-F. Rolland, M. Filali, M.-O. Reiser, D. Delanote, Y. Berbers, L. Pautet, and I. Perseil. UML & AADL '2007 Grand Challenges. *ACM SIGBED Review, A Special Report on UML & AADL Grand Challenges*, 4(4):(on line), october 2007.

[13] J.-P. Talpin, C. Brunette, T. Gautier, and A. Gamatié. Polychronous mode automata. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 83–92, New York, NY, USA, 2006. ACM.

[14] TOPCASED. Toolkit in open-source for critical apllications and systems development. http://www.topcased.org.

[15] S. Vestal. *MetaH User's Manual*. Honewell Technology Drive, 1998. http://www.htc.honeywell.com/metah/uguide.pdf.