



A Co-modeling Methodology for the Integration of Real-time Architecture Models

Isabelle Perseil, Laurent Pautet

GET-Télécom Paris

LTCI-UMR 5141 CNRS



ICECCS07- Auckland, New Zealand

Workshop « UML&AADL'2007 »



Overview

- Introduction
- Part 1 - Atomic components behavior and their accounting
- Part 2 - Using the AADL to model architectures
- Part 3 - Integrating the +CAL algorithm language into the AADL formalism
- Part 4 - A **proof approach** to ensure that the architecture requirements are followed
- Part 5 - A full **generated** application
- Conclusion

Multi layers of design

- Architectural configurations of real-time and embedded systems have to be formally verified
- Integration techniques of multiple domain-specific languages and tools
- Architecture design languages (ADLs)
- AADL

Algorithms and processes

- Non-functional properties are often specified through algorithms that mainly involve process units
- Processes are described separately from their behavior
- **Atomic operations** of an algorithm define what we call atomic component behavior
- These atomic operations are hidden in the state machine formalism, although they are **key elements** for proving the correctness of an algorithm

The grain of atomicity

- **Coarser-grained representation** (simpler)
 - ⇒ Sequence of system operations → a shorter sequence of steps (encapsulated)
 - ⇒ May fail to reveal important details of the system
- Or **Finer-grained representation** : more accurately describes the behavior
 - ⇒ Involves multiple suboperations
- ➔ **Choose the specification's level of abstraction ?**
 - ⇒ What system changes are represented as **a single step of behavior**

Semantic behavior of components

- The semantic behavior of AADL components is described **within an annex**
- Neither in the AADL standard, nor in its Behavioral Annex, is any language specified to describe atomic component behavior
- It could be expressed either in the most informal language (natural language) or in a very formal language

+CAL integration

- The integration of an algorithm language, +CAL, into an AADL specification is under construction through an annex mechanism
 - ⇒ It constitutes the heart of this present work
 - ⇒ together with the Ada code generation from Ocarina, our implementation of an AADL compiler

Overview

- Introduction - Algorithms are the heart
- **Part 1** - Atomic components behavior and their accounting
- **Part 2** - Using the AADL to model Architectures
- **Part 3** - Integrating the +CAL algorithm language into the AADL formalism
- **Part 4** - A proof approach to ensure that the architecture requirements are followed
- **Part 5** - A full generated application
- Conclusion

Atomic components behavior description

- Medvidovic's ADL classification leaves very little room for the atomic components behavior description
- From components to subcomponents, we have to ensure that the behavior is consistent, down to the finest grain
- What do we usually mean, in the architecture design language domain, by atomic component?

Services...

- Why an atomic element behavior has to be described within a totally different formalism?
- Atomic components are generally manipulated through a set of primitives that make up services
- Services are expanded but not reactively to the requirements changes; in addition, at a design level, we do not have a generic thread management

Choices...

- To avoid the dependence of a limited set of proved properties, we did not choose to build **a proved algorithms library**
 - ⇒ The analysis of a given algorithm **through a set of parameters** coming from the initial requirements
 - ⇒ **A fixed number of parameters / a suitable algorithm**
 - To prove
 - To guaranty that it is consistent with global architectural non functional properties

Context...

- To use a formal language to specify architectures in an industrial context ?
- A specific formal language, but not necessary linked to the semi formal architecture language
- Looking for a formal language that would be strongly linked to the architecture language
 - ⇒ allow ease of use and
 - ⇒ rich formal expressivity
- A gradual formal expressivity
- A progression in the formal expressivity can be obtained by language transformations
- From a language that we could integrate, we would also like to be able to generate formal verifications

How to handle a low level behavior design

- **Algorithm steps**
- A low level in terms of behavior would be the one in which every necessary element to figure out the processing is modeled
- **We need dynamic, functional, set elements and operators** to properly design the atomic components behavior
- We will refer to "**single steps**" by "**atomic component behavior**"

Survey of languages

- Natural language
- Controlled natural languages
- Pseudo code
- CSP-like languages

Choosing the suitable language for writing algorithms

- The right level of formalization ?
- From the usual pseudo-code, we will not have enough constraint formalism to generate a formal language
- Choosing the +CAL language brings numerous advantages
- The +CAL language provides
 - ⇒ the advantages of high-level code
 - ⇒ the precision of a formal language that can be mechanically checked
- TLA+ spec → PVS spec

Overview

- Introduction - Algorithms are the heart
- Part 1 - Atomic components behavior and their accounting
- **Part 2 - Using the AADL to model architectures**
- **Part 3 - Integrating the +CAL algorithm language into the AADL formalism**
- **Part 4 - A proof approach to ensure that the architecture requirements are followed**
- **Part 5 - A full generated application**
- **Conclusion**

Using the AADL to model architectures

- AADL relies on the notion of components
 - ⇒ Component interfaces / Component implementations
- An implementation of a thread or a subprogram can specify **call sequences** to other subprograms
 - ⇒ **execution flows** in the architecture
- Through the **use of properties** attached to AADL elements, AADL models can incorporate non-architectural elements:
 - ⇒ execution time
 - ⇒ memory footprint
 - ⇒ behavioral descriptions, etc
- Use AADL as a backbone to describe all the aspects of a system

Attaching behavior to AADL components

- The AADL does not particularly address the description of component behavior
- Using properties, source code can be associated with AADL components
 - ⇒ This allows for the production of executable applications from AADL descriptions

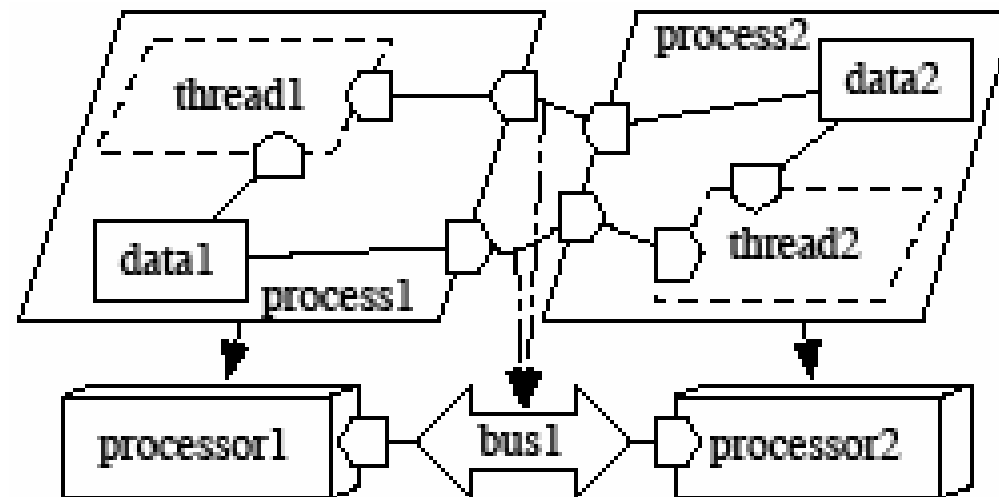
Attaching Behavior to AADL Components (2)

- The AADL standard also defines a behavioral annex to describe how to associate state machines to AADL components
- Describe the actions performed by the components of the architecture
 - ⇒ formal verification on the **execution of the components**
- Lists of states and transitions
 - ⇒ Transitions can depend on inputs, and can generate outputs or perform actions

Integration of Concurrent Behavior

- Behaviors involving several threads cannot be directly described
- The example on the next figure describes a situation in which describing local behavior, attached to each AADL thread, is not sufficient

Integration of Concurrent Behavior



Overview

- Introduction - Algorithms are the heart
- Part 1 - Atomic components behavior and their accounting
- Part 2 - Using the AADL to model architectures
- **Part 3 - Integrating the +CAL algorithm language into the AADL formalism**
- **Part 4 - A proof approach to ensure that the architecture requirements are followed**
- **Part 5 - A full generated application**
- Conclusion

Purpose

- The strict scope of an ADL
 - ⇒ Put the focus on describing the constraints on the architectural elements, with a property-based approach is an important step
 - ⇒ Constraints need a constraint language to be formally described
- For the MetaH language (from which AADL emerged), an accompanying language (ControlH) for modeling algorithms was developed
- concentrate on what and not on how ?
- represent the components separately from their behavior ?

Integration in AADL Models

- Where the most appropriate place to integrate algorithm structures is to be found ?
- Within the global system implementation ?
- This system implementation is the place where the main components (i.e. the processes, the processors, etc.) are instantiated and connected
 - ⇒ It is also the place to describe the way data are shared
- In order to be compliant with the AADL annex behavior specification, atomic behavior should also be attached to subprogram implementations

Modifying the Language itself or Adding an AADL Annex

- Two ways of undertaking an advanced algorithm language dedicated to describing component behavior in an AADL specification
 - ⇒ each element of AADL could take advantage of such a feature
 - ⇒ the algorithm language as one more AADL sublanguage

Annex subclauses types

system global

end global ;

system implementation global.i

Subcomponents

process1 : **process** a_process.i ;

process2 : **process** a_process.i ;

bus1 : **bus** a_bus ;

processor1 : **processor** a_processor ;

processor2 : **processor** a_processor ;

connections

[...]

annex algorithm specification { * *

algorithm my_algorithm

[...]

end algorithm

* * }

end global.i

Listing 1. Integration of an algorithmic annex in the architectural model

Choosing the suitable parameters

- Process synchronization design plays a central role in critical systems
- In order to illustrate the use of algorithms in a classical architecture design process, we show how a **mutex algorithm implementation** can have an influence on the resulting configuration
- This algorithm guarantees mutually exclusive access to a critical section among a number of competing processes
- At the same time, when two or more threads want to read or write the same memory area, we need a reliable mechanism to lock the access
- Among the well-known mechanisms: mutexes, semaphores, monitors and protected objects, we have kept the simplest, to avoid complicated design

Choosing the suitable parameters(2)

- Our intention is to describe a mutex algorithm with possible variants
- The variants would obviously correspond to requirements changes
- We show that these algorithms can't only be described into an aadl 'Property set'
- As a basis for argument, we will take a simple algorithm that is well-known
- Then, from the specification of this algorithm, we will show how we can check it using a translator to TLA+ and a model checker for a subclass of "executable" TLA+ specifications

Parameters

- At the AADL design level, we would like to be able to describe the parameters that will both have an influence on the construction of the algorithm and on the System architecture
- As starvation-free strictly depends on the chosen scheduling, either scheduling type is a parameter or the delay before entering the critical section
- Therefore, we may now determine around five relevant parameters
 - ⇒ delay parameter = delay entering in the critical section
 - ⇒ req performance parameters = number of memory access
 - ⇒ upper bound on time required to perform an atomic operation
 - ⇒ upper bound on time needed to execute the critical section
 - ⇒ time to stay in the critical section

Parameters (2)

- Another relevant parameter will be the **use of guards on the regions**, the implementation of conditional critical region changes the configuration
 - ⇒ This solution represents some drawbacks too, and so we may wish to change again the algorithm, writing the critical region as a procedure, encapsulated in a monitor, or a protected object

Lamport Bakery in +CAL

```
--algorithm bakery
variables Extraction = [ k \ i n 1 .. N | -> FALSE ] ,
Rank = [ m \ i n 1 .. N | -> 0 ] ;
process a_process \ i n 1 .. N
variable q ;
begin
  Extraction [ a_process ] := TRUE ;
  Rank [ a_process ] := 1 + max ( Rank [ 1 ] .. Rank [ N ] ) ;
  Extraction [ a_process ] := FALSE ;
  q := 1 ;
  while q /= N+1 do
    while ( Extraction [ q ] )
      do skip ;
    end while ;
    while ( ( Rank [ q ] /= 0 ) ∧ ( ( Rank [ q ] , q ) < ( Rank [ a_process ] , a_process ) ) )
      do skip ;
    end while ;
    q : q+1 ;
  end while ;
  \The critical section
  Rank [ a_process ] := 0 ;
  \ non-critical section ...
end process
end algorithm
```

Overview

- Introduction - Algorithms are the heart
- Part 1 - Atomic components behavior and their accounting
- Part 2 - Using the AADL to model architectures
- Part 3 - Integrating the +CAL algorithm language into the AADL formalism
- **Part 4 - A proof approach to ensure that the architecture requirements are followed**
- **Part 5 - A full generated application**
- **Conclusion**

A proof approach

- The +CAL language has many relevant features
- From +CAL, it is possible to generate as well a program into a procedural language like C++ (the c version), or Pascal (the p version), Java and Ada, as well as to automatically translate the +CAL specification into a language like TLA+
- Ex : Generating a TLA+ specification from the +CAL bakery algorithm
 - ⇒ First we reformat the +CAL algorithm as a TLA+ Module to be transformed by the translator

Generating a TLA+ specification from the +CAL bakery algorithm

- For the translation, we use the Lamport +CAL Translator
- Then, after translation, we obtain a TLA+ specification with labels
- The translation defines an action for each atomic operation of the algorithm

Reformatting the Bakery algorithm for TLA+ generation

```
----- MODULE Lamportbakery -----  
--algorithm LamportbakeryAlg  
variables Extraction = [ k \in 1..N | -> FALSE ] ;  
Rank = [ m \in 1..N | -> 0 ] ;  
process a process \in 1..N  
    variable q ;  
    begin  
        ....--> see +CAL algorithm (same)  
        cs : Rank [ a process ] := 0 ;  
    end process  
end algorithm  
\* BEGIN TRANSLATION  
\* END TRANSLATION  
=====
```

The Bakery algorithm in TLA+

VARIABLES Extraction , Rank , pc , q
vars == << Extraction, Rank , pc , q >>

ProcSet == (1 ..N)

Init == (Global variables)
 \wedge Extraction = [k \ i n 1 ..N \rightarrow **FALSE**]
 \wedge Rank = [m \ i n 1 ..N \rightarrow 0]
(* Process a_process *)
 \wedge q = [self \ i n 1 ..N \rightarrow {}]
 \wedge pc = [self \ i n ProcSet \rightarrow
CASE self \ i n 1 ..N \rightarrow " ncs "]

ncs (self) == \wedge pc [self] = " ncs "

\wedge Extraction' =
[Extraction **EXCEPT** ! [a_ process] = **TRUE**]
 \wedge Rank ' = [Rank **EXCEPT** ! [a_ process] =
1 + max(Rank [1] .. Rank [N])]
 \wedge pc ' = [pc **EXCEPT** ! [self] = "start"]
 \wedge **UNCHANGED** q

....

The Bakery algorithm in TLA+

.....

a_process (self) == ncs (self) \vee start (self)
 \vee lab1 (self) \vee lab2 (self)
 \vee lab3 (self)

Next == (\exists self \in 1 .. N: a_process (self))
 \vee (* Disjunct to prevent deadlock on termination *)
(\forall self \in ProcSet : pc [self] = "Done" \wedge **UNCHANGED** vars)

Spec == Init \wedge [] [Next] vars

Termination == $\langle \rangle$ (\forall self \in ProcSet : pc [self] = "Done ")


Overview

- Introduction - Algorithms are the heart
- Part 1 - Atomic components behavior and their accounting
- Part 2 - Using the AADL to model architectures
- Part 3 - Integrating the +CAL algorithm language into the AADL formalism
- Part 4 - A proof approach to ensure that the architecture requirements are followed
- **Part 5 - A full generated application**
- Conclusion

Generating Ada Code from the +CAL bakery algorithm

- From the previous +CAL bakery algorithm, we can easily **instantiate an Ada version**
- To be absolutely clear, we will only present the body of the algorithm
- In Ada, when the processes are waiting, we have the **delay** statement, to delay the execution for a specified period of time

The procedure entering of the Bakery algorithm in Ada



```
procedure entering (a_process : in (proc_index) is
begin
  Extraction(a_process) := true ;
  Rank(a_process) := 1 + maximum;
  Extraction(a_process) := false ;
  for q in 1 .. N loop
    loop
      delay 0 . 0 ;
      exit when not Extraction( q ) ;
      exit when (Rank ( q )=0)
        or (Rank (a_process)> (Rank ( q ) )
        or (a_process > q )
    end loop ;
  end loop ;
end entering ;
--
-- Exit Protocol
procedure way_out (a_process : in ( proc_index ) is
begin
  Rank (a_process) := 0;
end way_out ;
```

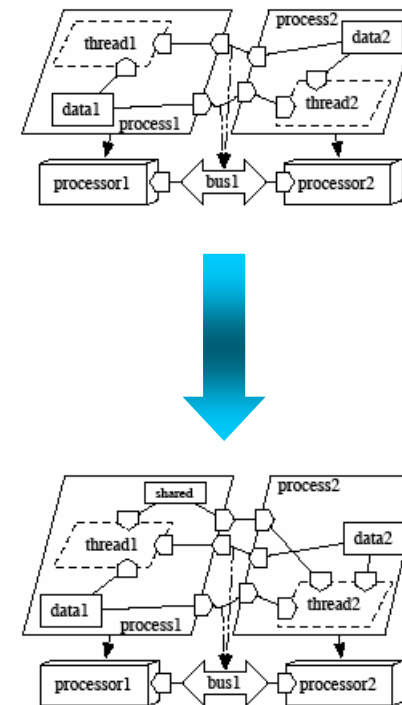
```
end algo Lamport bakery ;
```


Injecting the algorithm implementation into the AADL description

- From the +CAL description of the algorithm, we are able to produce source code
- We merge the source code we generate with the description of the initial architecture
- The implementation of the algorithm in itself implies some modifications in the code executed in the AADL threads
 - ⇒ to add calls to procedures such as entering
- In addition, the Bakery algorithm relies on two variables, shared by all the threads
- These variables have to be integrated into the architecture, as shown on the next figure
- The shared data is instantiated in one of the AADL processes, and accessed by all AADL threads

AADL(2) merging the source code we generate with the description of the initial architecture

- To add calls to procedures
- The Bakery algorithm relies on 2 variables, shared by all the threads
 - ⇒ have to be integrated into the architecture
 - ⇒ the shared data are instantiated in one of the AADL processes, and accessed by all AADL threads



The locking policy of the shared data is **centralized at the level of one process**,
→ can be easily managed

Introduction to a cross-checking procedure

- The **verification** on the Bakery algorithm obtained with TLA+ consists of a **proof-based approach**
- It is performed before the design of the actual architecture
- Once the architecture has been designed, it can be processed by tools such as Ocarina
 - ⇒ Thus we can perform model checking
- Prototypes of the modeled application can also be generated for tests

Overview

- Introduction - Algorithms are the heart
- Part 1 - Atomic components behavior and their accounting
- Part 2 - Using the AADL to model architectures
- Part 3 - Integrating the +CAL algorithm language into the AADL formalism
- Part 4 - A proof approach to ensure that the architecture requirements are followed
- Part 5 - A full generated application
- **Conclusion**

Conclusion (1)

- Architecture analysis and design is mostly performed without any standardized process or methodology
 - ⇒ very little traceability to handle the transition between the requirements, analysis and architecture design steps
- On the one hand, in describing the global requirements, the functional is separated from the non functional properties
- In the prototype phases, it is often necessary to adapt the algorithms to the architecture configuration, and vice versa

Conclusion (2)

- On the another hand, we build architectures that follow the requirements but, make abstraction of all the behavior constraints
- We have proposed to enhance the AADL language by providing an atomic component behavior design
- Our purpose is to **complete the existing gap between requirements and analysis**
 - ⇒ Our methodology provides a way, when choosing and updating parameters, to **dynamically build an optimal configuration**

Questions ?

