System Architecture Using SysML and AVATAR

Ludovic Apvrille
ludovic.apvrille@telecom-paristech.fr

Pierre de Saqui-Sannes
pdss@isae.fr

1st Summer School on Critical Embedded Systems
ISAE, Toulouse, France
July, 4th
About your Instructors

Ludovic Apvrille, Dr, HDR

- Associate Professor at Telecom ParisTech, Communication and Electronics Department, in Sophia-Antipolis, France
- Lecturer on Real-Time Operating Systems, and real-time UML and SysML
- Research focus: Embedded system partitioning, real-time UML and SysML, user-friendly formal verification, secured embedded systems
- Co-designer of the AVATAR modeling language
- Designer of the TTool tool, and prime developer

Pierre de Saqui-Sannes, Dr, HDR

- Professor at ISAE (Institut Supérieur de l’Aéronautique et de l’Espace), in Toulouse, France
- Academic advisor for student exchanges and focal point for Brazil
- Lecturer on real-time systems modeling (SysML, SCADE)
- Research focus: real-time SysML, protocol engineering, aeronautical systems
- Co-designer of the AVATAR modeling language
Goals

▶ To share an experience of real-time systems modeling
▶ To propose a language, a tool, and a method
  ▶ AVATAR, a modeling language based on SysML
  ▶ TTool for model simulation and user-friendly, but formal, verification
  ▶ A method that applies to a broad variety of real-time systems
▶ To exercise your modeling skills
  ▶ A copy of the TTool software for each attendant
  ▶ A real-time system all of you know: a microwave oven
▶ To discuss the pros and cons of using AVATAR and TTool
▶ To answer your questions
Agenda

09:00 - 12:00: Presentation of SysML and AVATAR
  ▶ Welcome
  ▶ An introduction to modeling
  ▶ An introduction to AVATAR and TTool
  ▶ Demonstration of the use of TTool on a "plate heater"

12:00 - 14:00 Lunch break

14:00 - 17:00: Lab session on TTool
  ▶ Modeling the subset of a drone system with TTool
    ▶ Sub-system: taking pictures at remotely defined GPS positions
  ▶ Simulation, verification and code generation from the model
Outline

1. UML, SysML and AVATAR
2. AVATAR Syntax
3. Model simulation
4. Patterns "Sensors - Controller - Actuators"
5. My First Avatar Model
6. Safety verification using UPPAAL
7. Security Verification
8. Rapid Prototyping
9. Conclusions
## About this Chapter

### Prerequisites
No prior knowledge of UML or SysML is requested

### Learning objective
Ability to understand the rationale behind the development of the AVATAR modeling language and the TTool tool

### Content
- Reminder on UML 1.5 and UML 2.0
- SysML
- AVATAR vs. SysML
UML 1.5

Requirement capture
Outside the UML model

Use case driven analysis
- Use case = main function
- System / environment (actors)
- Use cases need documentation

Object-oriented design
- Object = Name + Attributes (state) + Methods
- Objects communicate using method calls
- Software architecture defined by a class diagram
UML 2

Requirement capture
Outside the UML model

Use case driven analysis

Object-oriented design
- Object can communicate
  - by method calls
  - via ports
    - Input and output signals are defined by interfaces
What’s wrong with UML? (as far as systems modeling is concerned)

- Objects are for computer-literates, not for systems engineers
- Requirements are described outside the model using, e.g., IBM DOORS
- Allocation relations are not explicitly supported

Nevertheless SysML is a UML 2 profile

- Response to the UML for Systems Engineering RFP
- Developed by
  - The Object Management Group (OMG)
  - The International Council on Systems Engineering (INCOSE)

SysML standard:
www.omgsysml.org
An international standard at OMG (Object Management Group)

A graphical modelling language that supports the specification, analysis, design, verification, and validation of systems that include hardware, software, data, personnel, procedures, and facilities

A notation, not a method

Proprietary tools
  - Enterprise Architect, Rhapsody, Modelio, . . .

Free software tools
  - TOPCASED, Papyrus, TTool, . . .

User communities
  - http://sysmlfrance.blogspot.com/
  - http://sysmlbrasil.blogspot.fr/p/sysml-brasil.html
SysML Diagrams vs. UML Diagrams

- **SysML Diagram**
  - **Behavior Diagram**
  - **Requirement Diagram**
  - **Structure Diagram**
    - **Activity Diagram**
    - **Sequence Diagram**
    - **State Machine Diagram**
    - **Use Case Diagram**
    - **Block Definition Diagram**
    - **Internal Block Diagram**
      - **Parametric Diagram**
    - **Package Diagram**

- Same as UML 2
- Modified from UML 2
- New diagram type
From SysML to AVATAR

- AVATAR reuses most SysML diagrams
  - Requirement capture: requirement diagrams
  - Analysis: use case, sequence and activity diagrams
  - Design: block instances and state machines diagrams

- AVATAR does not entirely comply with the OMG-based SysML
  - In AVATAR, block instances diagrams merge block and internal block diagrams
  - AVATAR tunes SysML parametric diagrams to express properties (TEPE)
  - AVATAR does not support continuous flows

- AVATAR gives a formal semantics to several diagrams, including:
  - Block instance and state machine diagrams
    - Starting point for simulation, verification and code generation
Use of AVATAR Diagrams

Requirement capture → Requirement Diagram → Functions and Service Identification → Use-case diagrams

Property identification → Parametric Diagrams → Documentation (Scenarios + Flow-Charts) → Sequence diagrams + activity diagrams

Modeling Assumptions → TEPE Diagrams → Formal verification

Architectural design → Block Instances Diagram

Behavioral design → State Machine diagrams

Model Simulation
About this Chapter

Prerequisites
No prior knowledge of UML or SysML is requested

Learning objective
▶ Ability to read an AVATAR diagram
▶ Knowledge of the AVATAR syntax to be used during the lab

Content
▶ Requirements diagram
▶ Use case, sequence and activity diagrams
▶ Block instance and state machine diagrams
▶ Educational case study: a pressure controller
Pressure Controller

- **Specification** (from the client)
  - A pressure controller informs the crew when the pressure exceeds 20 bars. The alarm duration equals 60 seconds.
  - Two types of controllers. "Type 2" controllers keep track of the measured values.

- **System to be developed**: The controller

- **Modeling assumption linked to the system**
  - The controller’s set up procedure is not modeled
  - The controller’s shutdown procedure is not modeled
  - The controller never faces power cut problems
  - The controller’s maintenance is not modeled

- **Modeling assumptions linked to the system’s environment**
  - The pressure sensor will never fail
  - The alarm will never fail

- **Versioning**
  - The "keep track of measured value" option will not be modeled by the first version of the design diagrams
Requirement Node

- A requirement node identifies a requirement by:
  - A unique identifier (so as to achieve tracability)
  - A description in plain text (hopefully complying with a template)
  - A type (functional, non-functional, performance, security, ...).

Stereotype (UML extension mechanism)

The identifier, which is unique in the requirement diagram, enables traceability

<<Requirement>>
HighPressureDetection

ID=2
Text="The system shall check the cabin against high pressure."
Kind="Functional"

An informal text describes the requirement

Requirement type
- functional
- non functional
Relations Between Requirement Nodes

**Containment relation**
Splits up a compounded requirement into elementary ones

**Refinement**
Relates two requirements of different abstraction levels

**Derivation**
Builds a new requirement from the reuse of other requirements
Requirement Diagram - Pressure Controller

Shows how requirements are related to one another

1. **PressureController** (ID=0)
   - Text: "The system shall protect the crew against high pressure."  
   - Kind: "Functional"

2. **HighPressureDetection** (ID=2)
   - Text: "The system shall check the cabin against high pressure."  
   - Kind: "Functional"

3. **PDetection** (ID=3)
   - Text: "The system shall compare the pressure with a predefined threshold."  
   - Kind: "Functional"

4. **PressureSensor** (ID=4)
   - Text: "The system shall use a pressure sensor."  
   - Kind: "Functional"

5. **Information** (ID=5)
   - Text: "The system shall make the crew aware of danger."  
   - Kind: "Functional"

6. **Alarm** (ID=6)
   - Text: "The system shall monitor a sound alarm."  
   - Kind: "Functional"

7. **AlarmDuration** (ID=7)
   - Text: "The system shall monitor an alarm whose duration is equal to 60s."  
   - Kind: "Functional"

8. **Optional_Storage** (ID=1)
   - Text: "The system shall save the results of measurements."  
   - Kind: "Functional"

9. **RemovableDisk** (ID=8)
   - Text: "The system shall use a removable disk."  
   - Kind: "Functional"
Use Case Diagram

Use the following method:

1. Define the boundary of the system
   ▶ What is inside the rectangle is what you promise to realize
   ▶ What is outside the rectangle conveys a view of the system’s environment
     ▶ This is not part of your work!

2. Name the system

3. Separate the system from the external actors it interacts with

4. Identify the functions (services) to be offered by the system
   ▶ Not all the functions but those which interact with actors

Boundary
**Actors**

- **Syntax 1:** stickman

- **Syntax 2:** <<Actor>>

**Method**

- An actor identifier is a substantive
- An actor - or its descendants by inheritance relation(s) - must interact with the system
Use Case

- **Syntax**: "rugby ball" with exactly one use case

  ![Use case name]

Method

- A use case diagram is **NOT** an activity diagram
  - A use case describes a high-level function, not an elementary action
- A use case is described by a verb
  - The verb should convey the point of view of the system, not the point of view of the actors
Use Case to Use Case Relations

- **Inclusion**
  - A function mandatorily includes another function

- **Extension**
  - A function optionally includes another function

- **Inheritance**
  - A "child" function specializes a "parent" function
Use Case Diagram - Pressure Controller

- Shows what the system does and who uses it
Activity Diagram - Pressure Controller

- Shows functional flows in the form of succession of actions

```
act Initialize

act analyzePressure

[pressure>threshold]
[pressure<=threshold]

act informSupervisor
```
Sequence Diagram

- **An actor interacting with a system**

![Diagram of an actor interacting with a system]

- **Two interacting "parts" of the system**

![Diagram of two interacting subsystems]
Sequence Diagram - Communication Semantics

- **Synchronous communication** (black arrow)
  
  ![Diagram of synchronous communication]

- **Asynchronous communication** (regular arrow)
  
  ![Diagram of asynchronous communication]
Using Sequence Diagrams

Method

- A sequence diagram depicts one possible execution run, **NOT** the entire behavior of the system
- Inter-diagram coherence
  - A sequence diagram should not depict an actor or an instance block that does not appear on the use case diagram
Semantics

- One global clock (applies to the entire system)
- Time uniformly progresses (life lines are read top-down)
- Causal ordering of events on life-lines
  - Time information must be explicitly modeled

Relative dates

Absolute date
Timers

Set timer

Reset timer

Timer expiration

Sender

Receiver

{timer=myTimer, duration=10}

{timer=myTimer}

{timer=myTimer, duration=5}

{timer=myTimer}

{timer=myTimer}
Sequence Diagram - Pressure Controller

- Shows how the system and the actors communicate over time

PressureSensor

PressureController

Alarm

PressureSensor

PressureController

Alarm

pressure(16)
purpose(18)
purpose(18)
purpose(20)
startAlarm

{60..60}

stopAlarm
Block Instance Diagram: Syntax of Blocks

- A block name starts with an upper-case character
- An attribute name starts with a lower-case character
- A method name starts with a lower-case character
- A signal name starts with a lower-case character
- Compound names use upper cases as separators
Ports are connected to allow the state machines of blocks to exchange signals

A block instance may nest one or several block instances
State Machine - States and Transitions

Syntax

- State names may be written in upper or lowercase letters
- Attributes should be initialized

Method

- State names may be written in upper or lowercase letters
- Attributes should be initialized
A signal reception is a transition trigger

- The transition between INITIAL_STATE and END_STATE is triggered by a signal reception

- **FIFO queue communication**
  - The transition is fired if head(queue) = inputSignal

- **Synchronous communication**
  - The transition is fired whenever a rendezvous is possible

- Signals can convey parameters
The signal’s parameters, if any, are stored in attributes of the block instance that receives the signal.

- number : int;
- in signal(int param)

The signal reception uses a real parameter, which is an attribute of the block instance that contains the state machine.

The signal declaration contains a formal parameter.
From the same state it is possible to wait for several signals
- Asynchronous communication: the first signal in the queue triggers the transition
- Synchronous communication: The first ready-to-execute rendezvous triggers the transition
State Machine - Outputs

- A block instance can send signals with several parameters
  - TTool restriction: constants may not be used as real parameters; use attributes instead
  - A block instance cannot send two or several signals in parallel; it can send two or more signals in sequence

![Diagram of state machine with signals and attributes]
Signals declared by a block may be used by its sub-blocks.
Broadcast Channel

- All blocks ready to receive a signal sent over a broadcast channel receive it
- So, what happens if the channel of the following example is now set to broadcast?

```
<<block>>
~ in go_in()
~ out go_out()
~ in done_in(int x)
~ out done_out(int x)
<<block>>
T2
- x : int;
<<block>>
T1
- x : int;
<<block>>
T0
- x : int;
in done_in
in go_in
out done_out
out go_out
```

```
T0
  go_out()
  WaitForDone
  done_in(x)
T1
  go_in()
  x = 1
  done_out(x)
T2
  go_in()
  x = 2
  done_out(x)
```

L. Apvrille, P. de Saqui-Sannes (C)  System Architecture with AVATAR /TTool  July, 4th, 2013  41 / 114
A transition guard contains a *Boolean* expression built upon Boolean operators and attributes.
The *after* clause associates a \([\text{Tmin}, \text{Tmax}]\) interval to the transition’s enabling condition.

A transition with no *after* clause has de facto a *after*(0, 0) clause, which means the transition may be fired ”immediately”
State Machine Diagram - Pressure Controller

- Shows the inner functioning of the *Controller* block instance

![State Machine Diagram]

- **Running**
  - highPressure()
  - startAlarm()
  - after (alarmDuration, alarmDuration)
  - stopAlarm()
A timer must be declared as an attribute of the block instance which uses it

- Unlike attributes declarations, a timer declaration may not contain an initial value
  - Use the set operator to initialize the duration of a timer
- The signal issued by the timer at expiration time does not need to be declared
State Machines - Timers (2/3)

**Set**
- The "set" operation starts a timer with a value given as parameter
- The timescale is not specific to the "set" operation given the model uses a global clock that applies to all block instances

**Reset**
Prevents a previously set timer to send an expiration signal

**Expiration**
- The timer sends to the block instance it belongs to a signal whose name the timer’s name
- ⇒ a timer expiration is handled as a signal reception
"Temporally Limited acknowledgement" with timers

A block instance may take decisions depending on the signal which arrives first: either a "normal" signal or a timer expiration.
Simulation enables model debugging and therefore early detection of design errors in the life cycle of the system

**Driving the simulation**
- Step by step simulation
- "Random" simulation
- Breakpoints

**Tracing the simulation**
- Simulation trace in the form of a sequence diagram
- Each already visited branch within each state machine is clearly identified
- Attribute values may be displayed
Checking Design Diagrams against Syntax Errors

![Diagram of AVATAR Requirements and Design with syntax analysis and validation tools.]

- **AVATAR Requirements**
- **Analysis**
- **AVATAR Design**

**Validation**
- **Reviewed TClasses**
- **Syntax analysis / formal code generation:**
  -Warnings:
  -Errors:
- Actions - TURTLE gates
- **Invariants**

**AVATAR Block Diagram**

- **Controller**
- **ActuatorEmulator**
- **PSensorDriver**
- **PController**
- **PressureSensor**
- **Alarm**
- **SensorEmulator**

**Choosing blocks to validate**
- Blocks ignored
- Blocks taken into account:
  - Block0: Controller
  - Block0: PController
  - Block0: PSensorDriver
  - Block0: SensorEmulator
  - Block1: PressureSensor
  - Block1: Alarm
  - Block0: ActuatorEmulator

**Optimize specification**
- **Cancel**
- **Start Syntax Analysis**
Simulator Interface

Commands

Next fireable transition

Simulation Trace

Run simulation for x commands. Works only if the simulator is "ready"
Simulator Trace within a State Machine

Where the simulator has just stopped

Next fireable transition in the state machine

This state has been explored by the simulator

This transition has been explored by the simulator

Running

highPressure()

startAlarm()

stopAlarm()

after (alarmDuration,alarmDuration)
1. UML, SysML and AVATAR
2. AVATAR Syntax
3. Model simulation
4. Patterns ”Sensors - Controller - Actuators”
5. My First Avatar Model
6. Safety verification using UPPAAL
7. Security Verification
8. Rapid Prototyping
9. Conclusions
Modeling Assumptions

System
---------------
...
...
...

Environment
---------------
...
...
...

Language limitations
------------------------
...

Tool limitations
------------------------
...

Version 1
-----------
Date:
Author:
...

L. Apvrille, P. de Saqui-Sannes (C) System Architecture with AVATAR /TTool July, 4th, 2013 55 / 114
Requirement Diagram

GeneralRequirement
ID=0
Text="The system shall..."
Kind="Functional"

Inputs
ID=1
Text="The inputs shall..."
Kind="Functional"

Control
ID=2
Text="The control shall..."
Kind="Functional"

Outputs
ID=3
Text="The outputs shall..."
Kind="Functional"

UserInformation
ID=4
Text=""
Kind="Functional"
Use Case Diagram

System

Function_1
Function_2
Function_3

<<Actor>>
Sensor_1
<<Actor>>
Sensor_2
<<Actor>>
Sensor_m

<<Actor>>
Supervisor

<<Actor>>
Actuator_1
<<Actor>>
Actuator_2
<<Actor>>
Actuator_n
Simulation demands a Closed Model

Model with consistent diagrams

Controller

My use case

⇒

Sensor

Actuator

Model with a closed world (for simulation purpose)

SensorEmulator

Controller

ActuatorEmulator
Outline

1. UML, SysML and AVATAR
2. AVATAR Syntax
3. Model simulation
4. Patterns "Sensors - Controller - Actuators"
5. My First Avatar Model
6. Safety verification using UPPAAL
7. Security Verification
8. Rapid Prototyping
9. Conclusions
### About this Chapter

#### Prerequisites
- Use of a microwave oven
- AVATAR syntax

#### Learning objective
- Apply the AVATAR method to a system everybody should know
- Promote incremental modeling, starting from a very "simple" oven
- Exercise your modeling skills
- Use TTool’s model simulator

#### Content
- Specification: a "plate heater"
- Modeling assumptions, step-by-step modeling
- Simulation-based checking of the model against the requirements
Students can use a basic microwave oven at the university restaurant. The oven can basically only heat a plate for 30 seconds. The oven controller drives the magnetron. The user of the oven must first open the door, then put the plate inside, then close the door and finally start the oven. After 30 seconds, the oven stops and runs a bell to signal the user that the plate can be taken off. The oven comes with a rotating plate. An oven may also have a light turned on during cooking.

Un restaurant universitaire met à disposition de ses étudiants un four à micro-ondes rudimentaire doté d'une fonction unique : réchauffer une assiette pendant 30 secondes. Le contrôleur de four pilote un magnétron. L'utilisateur ou l’utilisatrice doit ouvrir la porte du four, déposer l’assiette à l'intérieur, fermer la porte et démarrer le four. Après 30 secondes le four s’arrête et émet un ”bip” pour avertir l’utilisateur ou l’utilisatrice qu’il/elle peut retirer l’assiette. Le four est équipé d’un plateau tournant. Certains fours ont en option une fonction qui maintient le four éclair lorsque le plateau tourne.
Um restaurante universitário oferece a seus estudantes um forno microondas rudimentar dotado de uma única função: aquecer um prato durante 30 segundos. O controlador do forno pilota um magnétron. O usuário deve abrir a porta do forno, colocar o prato no interior do forno, fechar a porta e partir o forno. Após 30 segundos, o forno pára e um “bip” avisa o usuário que ele pode retirar o prato. O forno é equipado com um prato girante. Alguns fornos têm uma opção: o forno é iluminado quando o prato gira.
More Practically...

Where to start?
Your instructors provide you with an AVATAR model. Please open it with TTool.

How to start? Read...
- Models of the “sensor controller actuators” patterns (to be applied to your model)
- Specification of the microwave oven
- The modeling assumptions we made to keep your model as much as possible simple

What to do next? Extend the model! But...
- Do not try to model the oven of your dreams
- keep to the specification i.e., your model must comply with the “plate heater” specification and that’s all we need for the lab
Modeling Assumption (1/2)

Boundary of the system to be modeled

- The system under design is the controller of the microwave oven
  - The controller will be detailed
  - The sensors and actuators will be emulated
    - Sensors output signals to the controller
    - Actuators accept all inputs from the controller

Assumptions about the system’s environment

- Sensors
  - The button used by the user will never fail
- Actuators
  - The magnetron will never fail
  - The bell will never fail
  - The tray will never fail
Assumptions about the oven

- The oven will never face a power cut
- The set up procedure will not be modeled
- The shutdown procedure will not be modeled
- The maintenance will not be modeled
- Design for testability is out of scope
Outline

1. UML, SysML and AVATAR
2. AVATAR Syntax
3. Model simulation
4. Patterns "Sensors - Controller - Actuators"
5. My First Avatar Model
6. Safety verification using UPPAAL
7. Security Verification
8. Rapid Prototyping
9. Conclusions
About this Chapter

Prerequisites
- AVATAR syntax
- Simulation of AVATAR models

Learning objective
- Check an AVATAR model against logical errors
- Check an AVATAR model against temporal violations

Content
- Brief reminder on timed automata and UPPAAL
- How TTool user-friendly conceals the use of UPPAAL
- Model of an extended version of the microwave oven
  - Suspending the magnetron when the door is opened in cooking mode
- Using observers
Simulation vs. Formal Verification

Simulation explores execution paths in the model relying on

- The experience of the Human who guides the simulation
- Random selection in case of non deterministic choice (several transitions fireable at the same time)

Formal Verification formally checks a model of the system against (a subset of) its expected properties

- Property expression using, e.g., logic formulas
- **Reachability analysis** with search through the state space of the system
  - TTool and UPPAAL (this slideshow)
- **Structural analysis** without state space exploration
  - TTool: invariants (not addressed in this slideshow)

Formal verification uses mathematics rather than chance
Properties

General properties that any system should verify

- From any state the system may return to its initial state
- Deadlock freeness
- No unspecified reception (in case of "blocking" queuing policy)
- No livelock (the system does not contain any piece of dead code)

Specific properties

E.g. "At any time, one station of the LAN holds the token."

Safety: Nothing bad will happen

E.g. "The microwave oven will not start heating as long as the door remains open."

Liveness: "Something good will eventually happen"

E.g. "All connections requests from the pilot will be acknowledged by the air traffic controller."
### Principle of reachability graph generation

1. From the initial state
2. Search for fireable transitions and create new states
3. Compare new states with existing ones
4. GOTO 2, and take newly created states as initial states

### Risk
- State explosion problem
- Missing resources (e.g. memory)

### (Some) Solutions
- State coding (hash functions)
- Partial exploration of the graph
Model Checking with UPPAAL

- Model
- Timed Automata
- Model Analysis
- TCTL formulas
- Timed Computation Tree Logic
- Properties
- Result
- Traces
Timed Automata Network

- Two-way synchronization on complementary actions

(a) Lamp.

(b) User.
TCTL (Times Computation Tree Logic)

E <> p

A [] p

E [] p
Transparent Use of UPPAAL from TTool

Formal link between the languages of TTool and UPPAAL

The formal semantics of AVATAR design diagrams (block instance and state machine diagrams) is given by translation to timed automata (a formal language that already has a formal semantics).

How TTool transforms an AVATAR model into UPPAAL automata

- The requirement, use case, sequence and activity diagrams are ignored
- TTool translates the block instance and state machines diagrams of the model into a timed automata network (UPPAAL format)

TTool makes the use of UPPAAL as much transparent as possible

- TTool invokes UPPAAL from an AVATAR design model
- TTool returns verification results at the AVATAR model level
  - That information is not intrusive
States that may be checked for reachability and liveness

Verification-Oriented Information in AVATAR Models

States that may be checked for reachability and liveness
Observer-Guided Verification

Basics of observers

- Expression of (complex) properties within the design
- Observer should have an *error* state whose reachability can be searched for in TTool/UPPAAL
- The observer should remain non-intrusive
  - At least, as long as the observed property is satisfied

Example: Pressure Controller

- Observer that verifies the alarm rings in zero time when a high pressure is detected
An "AlarmObserver" block is added to the design

AlarmObserver fetches information from the pressure sensor and the alarm
Whenever the observer gets a *highPressure* signal, it goes into the state ERROR after 1 unit of time if it hasn’t received yet an *alarm* signal.

The reachability of ERROR is searched for with UPPAAL.

The ERROR state is not reachable.
Outline

1. UML, SysML and AVATAR
2. AVATAR Syntax
3. Model simulation
4. Patterns “Sensors - Controller - Actuators”
5. My First Avatar Model
6. Safety verification using UPPAAL
7. Security Verification
8. Rapid Prototyping
9. Conclusions
About this Chapter

Prerequisites
▶ AVATAR syntax

Learning objective
▶ Model security mechanisms
▶ Model security properties
▶ Check an AVATAR model against security flaws

Content
▶ Brief reminder on Pi-calculus and ProVerif
▶ How TTool user-friendly interface conceals the use of ProVerif
▶ Application to an extended version of the microwave oven
   ▶ A remote control is introduced
   ▶ The communication between the remote control and the microwave must be secured
A few Limitations of AVATAR to Address Security

**Initial knowledge:** pre-sharing of cryptographic material (e.g., secret keys)

**Cryptographic functions:** No support for cryptographic functions, e.g., `encrypt-with-symmetric-key()`, `MAC()`, etc.

**Communication Architecture:** Can AVATAR channels be eavesdropped?

**Attacker model:** No attack model in AVATAR

**Security properties:** No facility to represent security properties
AVATAR Design Features for Security

Initial system knowledge: Knowledge shared by elements of the system for all system executions:

#InitialSystemKnowledge Alice.sk Bob.sk

Initial session knowledge: Knowledge shared by elements of the system for one system execution:

#InitialSessionKnowledge Alice.sk Bob.sk

Cryptographic functions: Predefined in each AVATAR block: MAC(), encrypt(), decrypt(), sign(), verifyMAC(), verifySign()...

Communication Architecture: public channels can be defined between blocks. Attackers can eavesdrop public channels (but not private ones)

Attacker model: Dolev-Yao. Taken from the underlying security framework ProVerif
Alice and Bob system

- Exchange of a confidential data
Use of cryptographic functions and data types

**Alice**

1. makingMessage
2. m.data = secretData
3. \( m_1 = sencrypt(m, sk) \)
4. sendingMessage
5. chout(m1)

**Bob**

1. waitingForMessage
2. chin(m2)
3. messageDecrypt
4. \( m = sdecrypt(m2, sk) \)
5. messageDecrypted
6. receivedData = m.data
7. SecretDataReceived
ProVerif: Main Features

- **Quite generic**: targets communicating systems modeling in general
- Well suited for **Communicating Entities (CEs)** modeling:
  - Based upon process algebras
  - CEs represented as pi-processes
- **Targets the proof of security properties**
  - Confidentiality
  - Authenticity
- **Dolev-Yao attacker model**
- **Completely automated verification**
# Syntax of Pi-Calculus supported by ProVerif

<table>
<thead>
<tr>
<th>M, N, a</th>
<th>terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>f(M₁...Mₙ)</td>
<td>function</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P, Q</th>
<th>processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>out(c,M); P</td>
<td>outputs ( M ) in ( c ) then ( P )</td>
</tr>
<tr>
<td>in(c,M); P</td>
<td>inputs ( M ) from ( c ) then ( P )</td>
</tr>
<tr>
<td>new a; P</td>
<td>defines ( a ) restricted to ( P )</td>
</tr>
<tr>
<td>event myEvt(x); P</td>
<td>executes an event ( myEvt(x) ) then ( P )</td>
</tr>
<tr>
<td>let x=g(M₁...Mₙ) in P</td>
<td>destructor application</td>
</tr>
<tr>
<td>else Q</td>
<td></td>
</tr>
<tr>
<td>if M=N then P else Q</td>
<td>conditional</td>
</tr>
<tr>
<td>P</td>
<td>Q</td>
</tr>
<tr>
<td>!P</td>
<td>infinite replication of process ( P )</td>
</tr>
<tr>
<td>0</td>
<td>null process</td>
</tr>
</tbody>
</table>
Formal Verification of Security Properties from TTool

States that ProVerif proved as reachable:

- The `secretData` attribute of Alice remains confidential.
- The m2 message of Bob received in state `messageDecrypted` is authentic.
Enhancing the Microwave with Security Mechanisms and Properties

Additional features

- The microwave oven now has a remote control to be remotely started
- A remote control is specific to one given oven, i.e., a remote control shall not be able to start another microwave
- No one, apart from the user of the remote control, shall know that the microwave oven was remotely started, and for which duration it was started

Work to do

- Extend the microwave model so as to support the remote control
- Add necessary confidentiality and authenticity properties
- Perform formal verification of your security properties with ProVerif
- Check that the safety properties are still verified
About this Chapter

Prerequisites
- AVATAR syntax
- AVATAR model simulation

Learning objective
- Generate executable code from an AVATAR model
- Execute generated code and analyze execution traces
- Add your own code to an AVATAR model

Content
- Overview of code generation in TTool
- Transformation of AVATAR design diagrams into executable code
- Application to an extended version of the microwave oven
  - Practice with code generation and execution
  - Adding your own code to an AVATAR model
Code Generation: Overview
Principle of Code Generation

- Only AVATAR design diagrams are taken into account
- Generated code relies on POSIX threads
  - One thread per block
- Synchronous communications between blocks is implemented in the AVATAR runtime with POSIX mutex
  - Asynchronous communications relies on linked lists managed in the AVATAR runtime
  - Time is handled based on POSIX `clock_gettime()` with `CLOCK_REALTIME` option
- ...
We now address the virtual prototyping scheme.
Virtual Prototyping Steps

1. Model refinement
2. Selection of an OS, setting of options of this OS (scheduling algorithm, . . .)
3. Selection of a hardware platform, and selection of a task allocation scheme
5. Manual code improvement - Code might also be manually added at model level
6. Code compilation and linkage with OS
7. Simulation platform boots the OS and executes the code
8. Execution analysis: directly in TTool (sequence diagram) or with debuggers (e.g., gdb)
Support for Virtual Prototyping: SoCLib and MutekH

Hardware platform simulator: SoCLib (www.soclib.fr)
- Virtual prototyping of complex Systems-on-Chip
- Supports several models of processors, buses, memories
  - Example of CPUs: MIPS, ARM, SPARC, Nios2, PowerPC
- Two sets of simulation models:
  - TLM = Transaction Level Modeling
  - CABA = Cycle Accurate Bit Accurate

Embedded Operating System: MutekH (www.mutekh.fr)
- Natively handles heterogeneous multiprocessor platforms
- POSIX threads support
- Note: any Operating System supporting POSIX threading and that can be compiled for SoCLib could be used
Virtual Prototyping: Graphical Environment

Main window of TTool

Console of MuteK

UML sequence diagram updated when simulating with SoCLib

SoCLib simulation based on a SystemC engine
(Virtual) Prototyping: Code Generation

Select options and then, click on 'start' to launch code generation / compilation / execution.
Virtual Prototyping: SocLib Simulation

SoCLib simulation based on a SystemC engine

Cycle Accurate System Simulator
ASIM/LIP6/UPMC
E-mail support: Richard.Buchmann@asim.lip6.fr
Contributors: Richard Buchmann, Sami Taktak,
Paul-Jerome Kingbo, Frederic P?trot,
Nicolas Pouillon

Last change: Dec 6 2011

Initializing memories with 5a
caba-vgm-mutekh_kernel_tutorial SoCLib simulator for MutekH
Initializing memories with 5a
Initializing memories with 5a
Virtual Prototyping: Console

Console of MutekH
(Virtual) Prototyping: Trace

TTool gathers execution traces and displays them in a sequence diagram

UML sequence diagram updated when simulating with SoCLib
Customizing Generated Code with Your Own Code

Application and block code

- Global code of the application
  - Inclusion of header files, global variables, ...
- Code global to one given block
Customizing Generated Code with Your Own Code (Cont.)

- Code can be provided for state entry action
  - i.e., action executed whenever a state is reached

States with entry code

Entry Code

Use of block variables

printf("Heating remaining time: %d\n", remainingTime);
Use of Customized Generated Code

Console debug
- Using e.g. `printf()` function

Connection to a graphical interface
- Piloting the code with a graphical interface
- Visualizing what’s happening in the executed code
- Connection to graphical interface via, e.g., `sockets`
Use of Customized Generated Code (Cont)

Graphical interface for the microwave oven

- Socket connection to a graphical interface programmed in Java
Requirements

- \textit{gcc} must be installed on your system
- You will execute the code on your local PC
  - Optionally, you may use SoCLib / \textit{MutekH}

Work to do

- Generate the code of your microwave oven
- Compile the code, execute it
- Analyze the execution trace
- Put your own code in the model, e.g., \textit{printf()}, and test
Outline

1. UML, SysML and AVATAR
2. AVATAR Syntax
3. Model simulation
4. Patterns ”Sensors - Controller - Actuators”
5. My First Avatar Model
6. Safety verification using UPPAAL
7. Security Verification
8. Rapid Prototyping
9. Conclusions
Conclusions

AVATAR

- SysML dialect for real-time systems modeling
- Formal semantics (design diagrams)

TTool

- Open-source toolkit available for Windows, MacOS, Linux

Applications

- Educational
- Industry
- Research
Ongoing Work

Search for mutual exclusion situations

- Does the microwave oven satisfy the following requirement: "the oven must not heat when the door is open"?
- Use of invariants

Protocol Engineering

- Tetrys protocol (design at ISAE)

Modeling of low-level software layers

- Security-oriented design of embedded systems
- Critical drivers of automotive systems
- ...
Incremental Modeling of a Microwave Oven

Version #1
- A plate heater that stops when the door is open
- Model simulation

Version #2
- A "plate heater" whose activity is suspended (resp. resumed) when the door is open (resp. "closed") while the controller is in the Cooking state
- Safety verification (reachability/liveness and observers)

Version #3
- A remote control device that securely communicates with the oven
- Security verification

Can we prove mutual exclusion between "Oven in Cooking state" and "Door in Open state"? → Use of invariants
References

TTool

- Download, install, documentation, examples: http://ttool.telecom-paristech.fr/

- Safety property verification

- Rapid prototyping

- Overview of the TTool tool

- Property expression in TEPE