Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems

John Hatcliff, Xinghua Deng, Matthew B. Dwyer, Georg Jung, Venkatesh Prasad Ranganath

Kansas State University

presented by Jaroslav Ševčík

Overview

- Goal – use formal methods in software development process
- Cadena – framework for
  - dependency analysis with varying levels of precision.
  - extracting checkable models from component specification.
- Based on CORBA component model
- Used for real-time systems
- Targetted at all mission/safety-critical systems

Formal methods

- Reasoning about various aspects of a program
- Uses mathematical logic for specifications
- Proving program invariants, preconditions, postconditions
- Types of formal methods
  - Light-weight – unsound and incomplete
  - Medium-weight – sound, but incomplete
  - Heavy-weight – sound and complete
- Successful applications in hardware
- Do not scale well enough for software

Automated Theorem Proving

- Heavy-weight technique
- Input: (annotated) program
- Verification condition generated:
  - from the code, e.g.
    - For `x.field = 0;` we have to prove that $x \neq 0$.
    - `a[i] = 0;` implies $i \leq a.length$.
  - from the annotations, i.e. class invariants, method pre- and post-conditions etc.
- Generate theorems using Hoare logic, weakest precondition etc.
- The verification condition then checked using an automated theorem prover.

Usage of ATP

- ACL2 theorem prover (parts of AMD K7 design)
- ESC/Java
  - unsound and incomplete
  - . . . e.g. loops, guessing object invariants, or axiomatization of floating point numbers
  - More like type checking, but uses heavy-weight tools
  - “How to shoot sparrows with cannons”
- Spec# - formal specification for C#
  - Similar to ESC/Java, but is sound and under some assumptions also complete
  - Powerful precondition, postcondition and invariant inference

Model checking

- Light-weight technique
- State-space exploration
- Based on a program abstraction (model)
  - Finite size
  - Reasonably large state space
- Usually used for
  - Protocols
  - Hardware
  - Concurrent programs

SPIN

- SPIN is the most widely used model checker
- Uses PROMELA language
- SPIN model consist of
  - type declarations
  - channel declarations
  - variable declarations
  - process declarations
- No unbounded data, channels or processes

Type declarations

- Basic types `bool, bit, byte, short, int` and arrays:

```
bool    flag = 1; /* declaration */
byte    counter[30];
flag = 0; /* assignment */
```

- Default value is always 0.
- Records:

```
typedef record {
  short f1;
  byte f2;
}
Record rr;
rr.f1 = ..;
```

Statements

- Two types of statements:
  - Executable – can be executed immediately
  - Blocking – cannot be executed
- Assignment is always executable
- Expression is also a statement
  - and is executable only if it evaluates to non-zero:
    - `2 < 3` always executable
    - `x < 27` executable if x smaller than 27
- `skip` statement is always executable
- `assert` statement is also always executable

Processes

Process declaration and creation:

```
proctype Foo(byte x)
{ ... }

init {
  int pid2 = run Foo(8);
}
```

Processes are:

- executed concurrently
- scheduled non-deterministically
- interleaved (no two statements execute at the same time)

Mutual exclusion

```
bool x, y, t;
proctype A()
{       x = 1;
        t = 1;
        (y == 0 || t == 0);
        /* critical section */
        x = 0 }
proctype B()
{       y = 1;
        t = 0;
        (x == 0 || t == 1);
        /* critical section */
        y = 0 }
init
{       run A(); run B() }
```

Assertions

- How to use SPIN to check the correctness for us?
- Use a variable that gets incremented and decremented in every critical section:

```
mutex++;
mutex--;
```

- . . . and run a process that will guard the variable:

```
active proctype monitor() {
  assert(mutex != 2);
}
```

- If the program is wrong the assert in the sequence

```
mutex++; /* <- P0 */ mutex++; /* <- P1 */
assert(mutex != 2); /* process monitor */
```

will fail!

Atomicity

- All statements are atomic
- `atomic{stmt1; stmt2; ...;stmtn}` statement can be used to define atomic sequences, e.g. atomic test and set:

```
atomic{ flag != 1; flag = 1 }
```

- No pure atomicity (even if statement i is blocked, all statements $1 \ldots i$ will be still executed).
- For deterministic sequences use `d_step` instead of `atomic`
  - No intermediate states are generated.
  - Reduces state space.
  - Possibly blocking or non-deterministic statements cause run-time errors!

Non-deterministic choice

- If statement:

```
int a = 0, x = 1, y = 1;
proctype A() {
    if
    :: x == 1 -> a = a + 1
    :: y == 1 -> a = a - 1
    fi }
```

- At the end $a = 1$ or $a = -1$, SPIN will make a non-deterministic choice.
- If $x \neq 1$ and $y \neq 1$, the statements will block.
- **do** statement similar to **if**
- Loops until an explicit **break** statement

Communication

- Channel declaration:

```
chan c = [2] of {bit}
```

- The number denotes the size of the message queue.
- Sending a message – using ! operator

```
c ! 1;
```

- If channel full $\Rightarrow$ the send statement will block
- Receiving a message:

```
c ? 1;
c ? x;
```

- If channel empty or the constant is not the same $\Rightarrow$ the receive statement will block

dSPIN

- Dynamic extension to SPIN
- Adds to the language
  - Memory management (**new** and **delete**)
  - Functions (**function**)
  - Pointers (**&** operator, type)
  - Function pointers (**ftype** type)
  - Local scoping
  - Garbage collection
- Simplifies object-oriented specification

Bandera

- Model-checking framework for Java
- Program states typically very large $\Rightarrow$
- Bandera uses
  - **Slicing** to eliminate the code that does not relate to the verified property
  - **Data abstraction** to decrease state size
    - E.g. vector can be often abstracted by a small set $\{ItemInVector, ItemNotInVector\}$.
- Its back-end supports various model-checkers (SMV, SPIN etc.)

Cadena architecture I

- Centered around CORBA middleware
- Components described using
  - CORBA IDL
    - Interfaces, data types and components
  - Cadena Assembly Description (CAD)
    - Static component allocation and configuration policy
  - Cadena Property Specification (CPS)
    - Dependencies and behaviors
  - Correctness Properties
    - . . . using Linear Temporal Logic

Cadena architecture II

- Outputs:
  - Generated code:
    - Stubs
    - Skeletons
    - Implementation (from IDL)
    - System assembly (from CAD)
  - dSPIN model
- Offers the possibility of graphical views
- . . . and spreadsheet views

Cadena methodology

Development steps:

1. Load library of components and their CPS
2. Define new components and CPS
3. Define CAD to specify connections between components
4. Examine dependencies using the graphical viewer
5. Attach non-functional aspect specification
6. Specify global correctness properties
7. Generate transition system model and model-check correctness properties
8. Revise system using feedback from analysis tools

Case study

- Avionics software based on Boeing Bold Stroke architecture
- The software manages
  - Cockpit displays
  - Navigation and tactical sensors
  - Weapon deployments
- Periodic and aperiodic processing
- Thousands of operating modes
- Technologies: CORBA, C++

Example

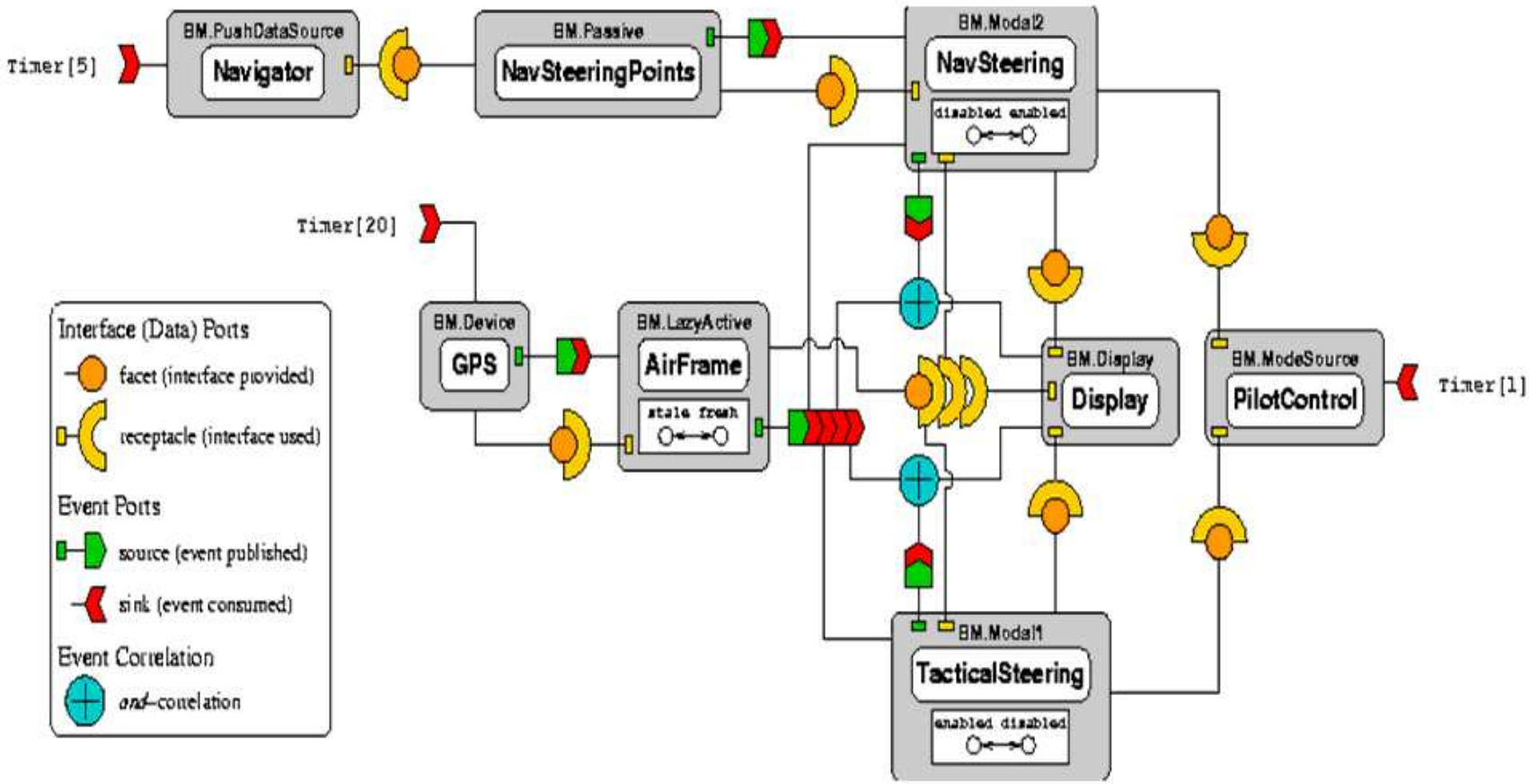
- Cadena system is demonstrated on a simple running example
- Simple avionics system that shows steering cues on a pilot's display
- Pilot can choose one of the modes:
  - *Tactical* display – mission objectives
  - *Navigation* display – navigation objectives
    - Derived from navigation points entered by navigator

Components

Components in the avionics system:

- `GPS` component, data sampled at a rate 20Hz
- The data passed to an intermediate modal component `AirFrame`
- `NavSteering` and `TacticalSteering` – modal components producing data for `Display` component based on data from the `AirFrame` component
- `Navigator` - polls for input at a rate 5Hz and pushes them to `NavSteeringPoints`
- `NavSteeringPoints` are used by the `NavSteering` component

Simple avionics system - diagram



Control-push data-pull

- Data transfered in two step process:
  1. Data producer publishes `DataAvailable` event
  2. Upon receiving the event data consumer calls *get data* accessor method
- Advantage – no thread blocking $\Rightarrow$ less opportunities for deadlock
- The `DataAvailable` notification sent through an event connection
- The data can be then retrieved through an interface connection

LazyActive component

- Example: `AirFrame` component
- `AirFrame` does not fetch data from `GPS` immediately
- ...it only sets `dataStatus` to `stale` and sends the `DataAvailable` event
- When then queried for data it checks the status
  - If `dataStatus = stale` it queries `GPS` for the data, sets status to `fresh` and returns the data
  - If `dataStatus = fresh` it just returns the (fresh) data

LazyActive component

```
module modalsp {
  interface ReadData {
    readonly attribute any data;
  };

  eventtype DataAvailable {};
  enum LazyActiveMode {stale, fresh};
  component LazyActive {
    provides    ReadData dataOut;
    uses        ReadData dataIn;
    publishes   DataAvailable outDataAvailable;
    consumes    DataAvailable inDataAvailable;
    attribute   LazyActiveMode dataStatus;
  };
};
```

Cadena Assembly Description

```
system ModalSPScenario {
  import cadena.common, cadena.modalsp;

  Rates 1, 5, 20;
  Locations 11, 12, 13;
  ...
  Instance AirFrame implements LazyActive on #LALoc {
   connect this.inDataAvailable
        to GPS.outDataAvailable runRate #LARate;
   connect this.dataIn to GPS.dataOut
   }
  Instance TacticalSteering implements Modal1 on 12 {
   connect this.inDataAvailable
        to AirFrame.outDataAvailable runRate 5;
   connect this.dataIn to AirFrame.dataOut;
  }
  ...
}
```

Dependency specification

```
component LazyActive {
  mode dataStatus;
  dependencydefault == none;

  dependencies {
    case dataStatus of {
      stale: inDataAvailable -> outDataAvailable;
             dataOut.get_data(); -> dataIn.getData();
      fresh: inDataAvailable -> outDataAvailable;
    }
  }
  behavior { ... }
  ...
}
```

Dependency analysis

- The steps for creating and refining dependency information:
  1. Use global dependence default (every input and output port can be connected)
  2. Specify dependency without modal behavior
  3. Refine by considering modes
  4. Refine by behavioral descriptions (which captures control-flow)
- Basic notions of dependency:
  - Inter-component dependencies – interface and event dependencies
  - Intra-component dependencies – trigger dependency

Dependency information usage

- Some properties can be devised automatically using the dependency information:
- Rate assignment to the ports
  - The rates can be distributed through the system
  - If more than one rate identified $\Rightarrow$ the higher one is used
- Distribution determination
- Synchronous dispatch optimization
  - Every non-correlated? co-located remote event delivery can be reduced to a synchronous call

Component modeling

- Straightforward translation from behaviors:

```
ftype Ref_NavSteering_dataIn1_getData;
ftype Ref_NavSteering_update;

function Fun_NavSteering_source1 (mtype t) {
  if
   :: NavSteering_componentState ==  enabled ->
    NavSteering_internalData = Ref_NavSteering_dataIn1_getData();
    Ref_NavSteering_update(NavSteering_DataAvailable)
   :: else
  fi
}
```

Modeling Middleware

- Event publication
  - . . . is achieved by placing function reference and event type to the channel thread process.
- Thread services
  - Rate monotonic scheduling:
  - Higher rate $\Rightarrow$ higher priority
  - Encoded using guarded actions
- Events modeled using rate groups and interfaces modeled using direct fuction calls?

Modeling Middleware

Modeling a rate group thread:

```
proctype RateGroup_1Hz () {
 ftype f;
 mtype m;
 ...
 do
  :: skip ->
   S_timeout?b;
   ...
   do
    :: Rate1_queue?[f, m] -> Rate1_queue?f(m); f(m)
    :: else -> break;
   od
 od
}
```

Thread services

```
function Fun_NavSteering_switch_getData () : int {
  atomic {
   P <= 1 ->
     P = MaxPriority
  }
  return NavSteering_componentState;
}
...
active proctype PriorityHandler() {
  do
   :: timeout -> d_step {
     if
      :: P > 0 -> P = P - 1;
      :: else
     fi
     }
  od
}
```

Simulating time

- No explicit notion of time
- Only the relative rate is preserved:
  - Take m as least common multiple of all rates in the system
  - Instantiate a counter t counting from 0 to $m - 1$ in a loop
  - Issue a timeout event for a rate k if $t \,\%\, m/k == 0$.
  - Can be scaled down by GCD of all the rates.
- Modeled using rendezvous channels in SPIN (i.e. channels with queue size 0).
- Each rate group has one channel and one SPIN process for the timeout events dispatch

Timer for 5, 10, 15Hz

```
proctype Timer() {      int t = 0;
  do
   :: (t >= 6) -> t = 0
   :: (t < 6) ->
    if
     :: (t \% 2) == 0 -> S_timeout15!1;
     :: else
    fi;
    if
     :: (t \% 3) ==  0 -> S_timeout10!1;
     :: else
    fi
    if
     :: (t \% 6) == 0 -> S_timeout5!1;
     :: else
    fi;
    t = t + 1
  od; }
```

Property Specification

- No explicit examples
- User can reason about:
  - call or return from a component method
  - publication or consumption of an event
  - values of component modes
- Observations can be qualified by
  - Component instance
  - Parameter and return values
  - Component ports
  - Rate groups
- Timeedit tool for visual editing of time properties

Experience

- Without enforcing the rate monotonic scheduling check aborted with
  - 26 million states
  - 1 GB RAM consumed
- With rate-monotonic scheduling:
  - 1.4 million states
  - 130 MB RAM
  - running time $\leq$ 1 minute.

Pros and Cons

Advantages:

- Formal guarantees of system properties
- Incremental dependency analysis

Disadvantages:

- Does not seem to scale well
- Non-functional modelling non-trivial (it is not clear has to be done manually and what is done by the tool)
- Oriented on very static systems
- Overall too tailored to Bold Stroke type systems