Towards checking Stateflow models with mCRL2

Jozef Hooman
jozef[dot]hooman[at]esi[dot]nl
Embedded Systems Institute (ESI)
Radboud University Nijmegen

joint work with
Remko van Cann & Jan Friso Groote (TU/e)

IPA Lentedagen on Integrating Formal Methods, 8 May 2008
Outline

- Context & motivation: TRADER project
  - Partners
  - Problem statement
  - Proposed outcome, partner contribution
- Stateflow model of TV
  - demo
- Checking models using mCRL2
  - Translation of Stateflow to mCRL2 language
  - Properties to be verified
  - Formalization of properties
  - Verification
- Concluding remarks
TRADER
System Reliability

20 fte/yr, 7 PhDs, 1 Postdoc, 10 Partners

Goal
Develop methods and tools to optimize reliability of high-volume products.

Issues
• Minimize product failures.
• Increase user satisfaction
  (user-centric design approach)

Carrying Industrial Partner
Previously known as Philips Semiconductors

Embedded Systems INSTITUTE

NXP

TU Delft

TU/e

University of Twente
The Netherlands

NXP

 imec

TU/e

TU/e technische universiteit eindhoven

Embedded Systems INSTITUTE

TASS software professionals

PHILIPS Consumer Electronics
Reliability threats in TV domain

Increasing complexity

• Functions/content increases rapidly
  – Play music (mp3 from usb), view photos, search teletext, Electronic Programming Guide, child lock, sleep timer, Picture-in-Picture, TV ratings, emergency alerts, many image processing options and user settings, …

• External information sources multiply
  – Connected planet strategy, downloadable applications

⇒ Increase of SW (1KB in 1980 – 24MB in 2007)
  Increase of third party content (EPG, codec’s)

Decreasing time-to-market

• Fixed shipping gates to occupy reserved shelf space

⇒ Faults in delivered products are a fact-of-life
Business impact

Not satisfying the high reliability expectations
- Many returned products
- Damages brand image
- Reduces market share

Cost of non-quality (CoNQ)
- 2-3% turnover (compare to research expenditures: 2.3%)
  Rudy Provoost (CEO Philips Consumer Electronics)

Challenge:
- **Prevent product faults causing customer complaints** given constraints:
  - Low costs
  - Short time to market
Trader – Proposed Outcome

• Methods and techniques that
  – Improve reliability *at development time*, e.g.,
    • By architectural improvements
    • By exposing product weaknesses that could lead to erroneous behavior
  – Improve reliability *after product release*, e.g.,
    • By giving the system *awareness that its customer-perceived behavior is or is likely to become erroneous*
    • By providing the system with a *strategy to correct itself in line with customer expectations*

• Supported by
  – *Proof of concepts & publications*
  – *Knowledge transfer to NXP and other industry*
Approach: run-time awareness

- Input
- System state
- System
- Awareness
- Compare model and system (output & state)
- Error
- Diagnosis
- Recovery
- Correction
- System output
Model-based Error Detection

How to decide whether system is correct as seen by user?

Use model of system behaviour!
Aim of TV Model

Aim is to make model of TV behaviour to:

• **Improve during development**
  support development by creating a single visual model instead of many distributed pieces of information

• **Improve after release**
  enable model-based error detection, introducing awareness
Focus of current TV model

Current modeling work concentrates on
• global control of TV
• user perceived behaviour;
  interaction via Remote Control and TV buttons

Ignore
• installation, settings
• image/sound quality
• screen size, view modes
• ambient light
• external devices
• internal modes & components
• …
Aim: model user-perceived behaviour of TV using executable state diagrams and visualize external IO.

Current tool support: Matlab/Simulink, mainly using Stateflow toolbox to define executable state diagrams with hierarchy, concurrency, data, events, …

Approach is rather tool-independent; diagrams are similar to state machines in UML-tools such as Rose RealTime and Rhapsody.

DEMO
Using the behavioural model (1)

Improve during development:

• to obtain concise, visual specification; currently spec is distributed over many documents

• to enable early detection of faults (e.g. ambiguities, omissions, inconsistencies, interference between features)

• to get quick feedback on product variations

• to generate test cases, to check conformance of implementations w.r.t. specification
Using the behavioural model (2)

Improve after release

Experiment with model-based error detection

- **Linux-based awareness framework** has been developed in which System Under Observation and code generated from model of its desired behaviour can be inserted easily

- **Open source media player MPlayer**
  model desired behaviour and insert system and spec in Linux-based framework

Note: for both applications of behavioural models, model correctness is important
Outline

• Context & motivation: TRADER project
  – Partners
  – Problem statement
  – Proposed outcome, partner contribution
• Stateflow model of TV
  – demo
• Checking models using mCRL
  – Translation of stateflow to mCRL language
  – Properties to be verified
  – Formalization of properties
  – Verification
• Concluding remarks
Approaches to improve model correctness:

• Simulation and visualization of external interface
• Check models using graphically specified properties, e.g. using
  – Reactis, also allows generation of test scripts to check conformance after model changes
  – Simulink Design Verifier

Question:

• Can we use the mCRL2 toolset to detect errors in Stateflow models?

Main case studies:

• Small Sound example
• Larger TV model
Sound example (1)
Sound example (2)
Sound example (3)

OnScreenDisplay

MuteOSD

clk[t-lastmuteentry >= DisplayMax]

ncDisplay
entry: display=0;

Mute

on
entry: display=1;

DisplayUnMute

clk[t-firstmuteentry>=DisplayMin]

uncertain

PossibleDisplay

display entry: display=2;

LastUnmute/lastmuteentry=t;

LastUnmute/lastmuteentry=t;

clk[t-firstmuteentry>=DisplayMin]

awaitLast

FirstUnmute/firstmuteentry=t;
Observe that Stateflow semantics is deterministic; each external event leads to a precisely defined sequential sequence of actions.

Many possibilities to translate Stateflow models to mCRL2, e.g.:

- Parallel states $\approx$ mCRL processes
- Translate event-based communication to synchronous communication with buffers
- ...

After some experiments, we translate each state $S$ to a process which executes an event:

$$S\_exec(\text{event:Events})$$

It calls other processes for entry and exit of states.
sort Events = struct
    % Simulink events
    Clk?isClk | 
    VolumeUp?isVolumeUp | 
    VolumeDown?isVolumeDown | 
    MuteOnOff?isMuteOnOff | 

    % Stateflow events
    EMute?isEMute | 
    FirstUnmute?isFirstUnmute | 
    LastUnmute?isLastUnmute ;
Translation to mCRL2 (3)

\[
\text{sort } \text{ListInt} = \text{List(Int)}; \\
\text{sort } V = \text{struct} \\
\quad \text{SUO} \mid \text{TV} \mid \text{ProcessKeys} \mid \text{Active} \mid \% \text{ state} \\
\quad \text{ActiveAudio} \mid \AAOn \mid \text{Mute} \mid \% \text{ state} \\
\quad \text{OnScreenDisplay} \mid \text{MuteOSD} \mid \% \text{ state} \\
\quad \text{DisplayUnMute} \mid \% \text{ state} \\
\quad \text{volume} \mid \text{vlevel} \mid \text{display} \mid \% \text{ data objects} \\
\quad \text{Delay} \mid \text{FME} \mid \text{LME} \mid \% \text{ delays} \\
\]

\[
\text{proc } \text{data}(d:\text{ListInt}) = \\
\quad \sum v:V. \text{sRead}(v, d.\text{var2nat}(v)) . \text{data}(d) \\
+ \sum v:V, m:\text{Int}. (\text{var2nat}(v) < (#d)) -> ( \\
\quad \text{rWrite}(v, m) . \text{data}(\text{writeVar(\text{var2nat}(v), m, d, [])}) ) ;
\]

Number of active substate
(0 id none, highest number for parallel states)
% Stateflow process
proc  stateflow =
  start(Clk).SUO_exec(Clk)
  . sum d:Int.
    rRead(Delay,d)
    . ( (d>0)->( sWrite(Delay,d-1 )<>(intern) )
    )
  . sum d:Int.
    rRead(FME,d)
    . ( (d>0)->( sWrite(FME,d-1 )<>(intern) )
    )
  . sum d:Int.
    rRead(LME,d)
    . ( (d>0)->( sWrite(LME,d-1 )<>(intern) )
    )
  . stateflow ;
% Execution processes
Proc  TV_exec(event:Events)=
  sum state:Int.rRead(TV,state).( 
    (state == 0)->( 
      intern )
  + (state >  0)->( 
      ProcessKeys_exec(event) % State: ProcessKeys 
      . Active_exec(event)   % State: Active     ) );
Translation to mCRL2 (6)

proc ProcessKeys_exec(event:Events)=
  sum state:Int.rRead(ProcessKeys,state).( 
    (state == 0)->( intern ) 
  + (state == 1)->( % State: awaitKey 
    sum d:Int.rRead(Delay,d).( 
      ( isClk(event) && d==0 )->( 
        ProcessKeys_exit 
        . ml_buffer(1) 
        . ( intern 
          + start(MuteOnOff).SUO_exec(MuteOnOff) 
          + start(VolumeUp).SUO_exec(VolumeUp) 
          + start(VolumeDown).SUO_exec(VolumeDown) 
        
        . ProcessKeys_enter([1]) 
      )<>(( isClk(event) && ( d > 0 ) )->( 
        ProcessKeys_exit 
        . ProcessKeys_enter([2]) 
      )<>( intern )) ) ) ) 
  + (state == 2)->( % State: Delay 
    sum d:Int.rRead(Delay,d).( isClk(event) && ( d == 0 ) )->( 
      ProcessKeys_exit 
      . ProcessKeys_enter([1]) 
    )<>( intern ) ) );
% Entry processes
proc ActiveAudio_enter(In:ListInt)=
    sum n:Nat.rRead(Active,n).(n < 1)->(
        Active_enter(1 |> In)
    )<>( (#In == 0)->(
        intern.ActiveAudio_enter([1])
    )<>
    ( (head(In) == 1 )->( % State: AAOn
        sWrite(ActiveAudio,1)
        . AAOn_enter(tail(In))
    )
    + (head(In) == 2 )->( % State: Mute
        sWrite(ActiveAudio,2)
        . Mute_enter(tail(In))
    )
    ));

For parallel states, all substates have to be entered, starting from lowest number.
% Exit processes
proc Active_exit =
    sum state: Nat.rRead(Active,state).( (state == 0)->( intern )
    + (state == 1 )->( %State: ActiveAudio
        ActiveAudio_exit )
    + (state == 2 )->( %State: OnScreenDisplay
        OnScreenDisplay_exit )
    . sum new: Nat.rRead(Active,new).(new == state)->( sWrite(Active,state-1)
        . (1<state)->(Active_exit)<>(intern)
    )<>( intern ) ;

Parallel state, exit all substates, starting from highest number
Requirements specification

R01: The initial state of the system is unmuted
R02: Volume is never less than MinVol or greater than MaxVol
R03: The MuteOnOff button will cause a transition to state muted
R04: The VolumeUp and VolumeDown button will not change the state to muted
R05: If volume is less than MaxVol, the VolumeUp button increases volume
R06: If volume is greater than MinVol, the VolumeDown button decreases volume
R07: When the system becomes muted, volume is eventually set to zero and it is not changed until the system becomes unmuted
R08: When the system becomes muted, muteMessage is eventually shown and it is not removed until the system becomes unmuted
R09: The VolumeUp, VolumeDown and MuteOnOff buttons will cause a transition to unmuted
R10: When the system becomes unmuted, unmuteMessage is eventually shown, after a finite time the message is removed
R11: When the system becomes unmuted and the transition is caused by the MuteOnOff button, the volume is eventually set to the original value of the unmuted system
R12: When the system becomes unmuted and the transition is caused by the VolumeUp button, the volume is eventually set to the increased original value of the unmuted system
R13: When the system becomes unmuted and the transition is caused by the VolumeDown button, the volume is eventually set to the decreased original value of the unmuted system
Requirements formalization R8

R08: When the system becomes muted, muteMessage is eventually shown

\[
( \text{true} \times . \text{cWrite(ActiveAudio, 2)})^\ast \\
( \mu X. (\text{true} \times \text{true} \\
\quad \text{and} [\text{not } \text{cWrite(display, 1)}])^\ast X )
\]

and it is not removed until the system becomes unmuted

\[
\text{Enter mute} \\
\text{Eventually display mute message}
\]

\[
\text{Enter mute state} \\
\text{Display mute message} \\
\text{No exit of mute state} \\
\text{Display changed} \\
\text{The sequence above should NOT occur}
\]
Requirements formalization R11

R11: When the system becomes unmuted and the transition is caused by the MuteOnOff button, the volume is eventually set to the original value of the unmuted system

\[
\begin{align*}
( [ \text{true}^* & \begin{align*}
& \text{"cWrite(ActiveAudio, 1)"} \\
& ( \text{not } \text{"cWrite(ActiveAudio, 0)"} )^* \\
& \text{"cWrite(volume, 0)"} \\
& ( \text{not } \text{"cWrite(ActiveAudio, 0)"} )^* \\
& \text{"start(MuteOnOff)"} \\
& ( \text{not } \text{"cWrite(ActiveAudio, 2)"} )^* \\
& \text{"cWrite(ActiveAudio, 2)"} \\
& (\text{not ( } \text{"start(MuteOnOff)"} \\
& \quad \text{ or } \text{"start(VolumeUp)"} \\
& \quad \text{ or } \text{"start(VolumeDown)"} \\
& \quad ) \text{)}^* \\
& \text{"start(MuteOnOff)"} \\
& \text{mu X. (true}\text{true} \\
& \quad \text{and } [\text{not } \text{"cWrite(volume, 0)""]}) \\
\end{align*}
\right) \text{and}
\end{align*}
\]

< true^* 
  \begin{align*}
& \text{"cWrite(ActiveAudio, 1)"} \\
& ( \text{not } \text{"cWrite(ActiveAudio, 0)"} )^* \\
& \text{"cWrite(volume, 0)"} \\
& ( \text{not } \text{"cWrite(ActiveAudio, 0)"} )^* \\
& \text{"start(MuteOnOff)"} \\
& ( \text{not } \text{"cWrite(ActiveAudio, 2)"} )^* \\
& \text{"cWrite(ActiveAudio, 2)"} \\
& (\text{not ( } \text{"start(MuteOnOff)"} \\
& \quad \text{ or } \text{"start(VolumeUp)"} \\
& \quad \text{ or } \text{"start(VolumeDown)"} \\
& \quad ) \text{)}^* \\
& \text{"start(MuteOnOff)"} \\
& > \text{true}
\end{align*}

To check existence of at least one path
Verification (1)

- Revealed errors in requirements formalization
- After fixing these errors, all properties hold

But not all combinations of timing parameters were checked

If \texttt{UnmuteDelayMax = UnmuteDelayMin} we get delay 0, and can be interrupted by \texttt{VolumeUp} before \texttt{LastUnmute} may occur
**Verification (2)**

**ActiveAudio**

**On**
- UnMute
  - entry: volume = vlevel;
  - clk[delay == 0]
- VolChangeDelay
  - clk[delay == 0]/LastUnmute;
- PossibleMute
  - clk[delay == 0]/FirstUnmute; delay = UnmuteDelayMax - UnmuteDelayMin;
- ShortestUnmute
  - /delay = UnmuteDelayMin
- MuteOnOff/delay = MuteDelayMax;

**Mute**
- MuteDelay
  - exit: volume = 0; Mute;
  - clk[delay == 0]
- MuteState

**SPEC:** lower and upperbound on mute delay

- VolumeUp[vlevel < MaxVol]/vlevel++;
- VolumeDown[vlevel > 0]/vlevel--;
- /delay = VolDelayMax;
- VolumeUp[vlevel >= MaxVol];
- VolumeUp[vlevel <= MaxVol];
- VolumeDown[vlevel <= 0];
Verification (3)

Note: if $\text{DisplayMin} \leq \text{UnmuteDelayMax} - \text{UnmuteDelayMin}$
then no mute display
Current attempt is to use “sum” in mCRL2 to include all possible values of $\text{DisplayMin}$
Verification of full TV model (1)

- State space explosion due to small error in definition of modulo function; problem was difficult to find
- After correcting this error, state space is approximately 10 million states
- Checked requirements R1 – R11

- R09 (*The VolumeUp, VolumeDown and MuteOnOff buttons will cause a transition to unmuted*) does not hold – this is not a requirement for this model
Verification of full TV model (2)

R11 (When the system becomes unmuted and the transition is caused by the MuteOnOff button, the volume is eventually set to the original value of the unmuted system)

→ Should not hold, but was valid for first version of formalization which was based on Sound model; this spec explicitly forbids VolumeUp & VolumeDown

( [ true*
    . "cWrite(ActiveAudio, 1)"
    ...
      . "cWrite(ActiveAudio, 2)"
      . (not ( "start(MuteOnOff)"
           or "start(VolumeUp)"
           or "start(VolumeDown)"
           ) )*
      . "start(MuteOnOff)" ]
( mu X.( <true>true
    and [not "cWrite(volume, 0)"] )
))and( After changing formalization property did not hold
R08 (When the system becomes muted, muteMessage is eventually shown and it is not removed until the system becomes unmuted)

Does not hold due to the order of internal steps in TV model:

- first internal Mute event is generated which leads to display of mute message;
- next mute state is entered and volume is set to zero
Concluding remarks

- Translation stable, only part of Stateflow; could be extended, optimized and automated
- Verification seems feasible; more work is needed to investigate scalability
- Requirements are difficult to formalize:
  - Difficult to read and to assess correctness
  - Current properties dependent very much on internal states and events and, hence, sensitive to model changes
  - Extending “forall paths” properties with (redundant) checks on existence of path useful
  - No clear method (yet) for formulating properties on stable states (where the model waits for external input)
- Requirements definition might be improved using graphical representation, e.g. defining observers in Stateflow
- Visual feedback of error trace in terms of Stateflow simulation would be convenient

Thank you for your attention!