42: Programmable Models of Computation for a Component-Based Approach to Heterogeneous Embedded Systems

Florence Maraninchi
Tayeb Bouhadiba

Verimag

26-30 November 2007
Contents

1 Introduction
2 Related work
3 42 basics
4 Discussions
5 42 Protocols
6 Comments
Contents

1. Introduction
2. Related work
3. 42 basics
4. Discussions
5. 42 Protocols
6. Comments
Embedded systems are heterogeneous:
Hardware/Software/OS, Synchronous/Asynchronous, Continuous/Discrete, ...

Heterogeneity in the design flow:
Several levels of abstraction (RTL,TLM,...), Several notions of time and data granularity, Several stages of development,...
42 : programmable models of computation for component based approach to **embedded systems**

Embedded systems are heterogeneous:
Hardware/ Software / OS, Synchronous / Asynchronous, Continuous / Discrete, ...

Heterogeneity in the design flow:
Several levels of abstraction (RTL,TLM,...), Several notions of time and data granularity, Several stages of development,...

⇒ Virtual prototyping and model driven development.
Introduction

42 : programmable models of computation for component based approach to embedded systems

Components are everywhere. (Hardware and software).

**HW :** (IPs) Of-the-shelf hardware components, available as specification or hardware blocks.

**SW :** EJB, .NET, CCM, Fractal, Sofa,...
Component based approach to embedded systems

Components are everywhere. (Hardware and software).

**HW**: (IPs) Of-the-shelf hardware components, available as specification or hardware blocks.

**SW**: EJB, .NET, CCM, Fractal, Sofa,…

Synchronous semantics (RTL).
Not adapted for our case.
42 : programmable models of computation for component based approach to embedded systems

MoCCs : The way components are activated and communicate together.

Several models of computation and communication should be taken into account. (SDF, DE, SR,...) Ptolemy.
Introduction

42 : programmable models of computation for component based approach to embedded systems

MoCCs : The way components are activated and communicate together.

Several models of computation and communication should be taken into account. (SDF, DE, SR,...) Ptolemy.

Semantics of heterogeneous components → MoCC definition
42 : programmable models of computation for component based approach to embedded systems

42 is a formalism which requires a special representation of components. It allows reasoning on components at the system level.

We focus on the assembly checking and allow programming several MoCCs in the same setting by means of basic primitives.
A typical embedded system
A typical embedded system

On a processor:
- A real-time OS +
- several processes

Each process can be programmed in a component-based framework

Hardware IP’s (components)
Contents

1. Introduction
2. Related work
3. 42 basics
4. Discussions
5. 42 Protocols
6. Comments
Around the formalization of MoCCs

Ptolemy
Directors implement the different MoCCs

Rialto
Hierarchical blocks of programs. A policy defines the MoCC at each level.

SML-sys (≠ SysML)
Functional programming framework of heterogeneous MoCCs

MoCCs are Programmable
Modeling in Lustre

- An example modeling two processors running a scheduler with two threads.
- Components and the global system are modeled using Lustre
- Asynchrony is modeled using additional signals

Virtual execution of AADL Models via translation into Synchronous Programs, EMSOFT’07
Ptolemy

A set of actors (Components) Assembled with wires + Director which defines the MoCCs
Contents

1. Introduction
2. Related work
3. 42 basics
4. Discussions
5. 42 Protocols
6. Comments
42 motivations

- A formal model that allows reasoning on components at the system level, definition of components, assembling components and checking the assemblage without looking at how components are implemented.

- Allowing expression of multiple MoCCs so that heterogeneity of embedded systems would be taken into account.

- “Forget As Much as Possible As Soon As Possible”
Restricting the context

+ Discrete systems with inputs and outputs
  (digital hardware components, reactive software pieces, simulated models of continuous physical environments)
- continuous systems
- non-oriented systems
+ Functional system-level descriptions
- extra-functional properties
  (energy consumption, temporal performances, ...)

MARANINCHI & BOUHADIBA (Verimag)
Main ideas(1) A basic Component
Main ideas(1) A basic Component

For ic1:

```java
a = id1.get();
b = id2.get();
od1.set(f(a, b));
oc1.set(false);
....
```

Data Ports

Input

Control Ports

Output

Data Ports

Input

Control Ports

Output
Main ideas(2) A complete System

Comp A

Comp B

Comp C

Comp D

ic, oc
input port
output port

\( u \)

\( v \)

\( w \)

\( z \)
Main ideas(2) A complete System

- Dialogs with components
- Manages memories associated to the wires
- Defines global ports
- No semantics until MoCC is chosen

\[ ic, oc \]

- input port
- output port

\[ Comp \]

\[ Comp A \]

\[ Comp B \]

\[ Comp C \]

\[ Comp D \]
Main ideas(2) A complete System

No semantics until MoCC is chosen
Main ideas(2) A complete System

Controller

Comp A

Comp B

Comp C

Comp D

ic, oc

input port

output port

MARANINCHI & BOUHADIBA (Verimag)
Main ideas(2) A complete System

Dialogs with components
Main ideas(2) A complete System

Dialogs with components
Manages memories associated to the wires
Main ideas(2) A complete System

Dialogs with components
Manages memories associated to the wires
Main ideas(2) A complete System

Dialogs with components
Manages memories associated to the wires
Defines global ports

Controller

Comp A
Comp B
Comp C
Comp D

m_d  m_c  m_b  m_a

x
y
o1
i1

Comp A
Comp B
Comp C
Comp D

ic, oc
input port
output port

MARANINCHI & BOUHADIBA (Verimag)
Main ideas (3) The controller

Controller is:

```java
var M : bool = true ;
for X do:{ /* defines X.
    m_a, m_b, m_c: FIFO(1,int);
    m_d, m_e, m_f: FIFO(4,int);
    if (M) {
        m_a.put ; /* reads i1.
        m_a.get ; D.z;//*activates D via z.
        m_f.put ; m_f.get ;
        A.u; m_b.put; m_d.put;
        m_b.get; B.v; M = M or p ;
        m_c.put ; m_c.get ;/*defines o1.
        m_d.get ; C.w ; m_e.put ;
        m_e.get; D.k ;
        M = ! M ;
    } else { ... }
    y = M; /*defines y.
}
```

- Dialogs with components
- Manages wires memories
- Defines global ports
Summary

- Only data is communicated between components
- The architecture describes how data flows
- Only the controller interacts with components via control ports
- The controller manages the memory associated with the wires
- Lifetime of this memory is limited to a macro-step
- \( \Rightarrow \) The MoCC is defined by the controller
Contents

1 Introduction
2 Related work
3 42 basics
4 Discussions
5 42 Protocols
6 Comments
Discussions (1) Basic components (Leaves)

42-ization of components

No restrictions on how components are made

Reuse of existing code

Definition of data and control ports
Discussions (1) Basic components (Leaves)

42-ization of components

No restrictions on how components are made

Reuse of existing code

Definition of data and control ports
Discussions (1) Basic components
(Leaves)

Scade diagram (Lustre)

\[ \text{C code generation} \]

\[
\text{void } \text{C1}\_\text{reset}(\text{outC}_C1^* \text{ outC})
\]

\[
\text{void } \text{C1}(\text{inC}_C1^* \text{ inC}, \text{ outC}_C1^* \text{ outC})
\]

42-ization of components

No restrictions on how components are made

Reuse of existing code

Definition of data and control ports
Discussions (1) Basic components (Leaves)

Scade diagram (Lustre)

C code generation

void C1_reset(outC_C1 * outC)

void C1(inC_C1 * inC, outC_C1 * outC)

Piece of code

42-ization of components

No restrictions on how components are made

Reuse of existing code

Definition of data and control ports
Discussions (2) : The controller

Memory associated to wires

Controller

\[
\begin{align*}
M_a \\
M_b \\
\ldots
\end{align*}
\]
Discussions (2) : The controller

Controller output $\in L = \{(m_i.get + m_i.set + C_i.lc_i)^*\}$

$m_i$ : The set of memories associated with the wires

$C_i.lc_i$ : The union of all Components’ input control
Discussions (2) : The controller

Controller output \( \in L = \{(m_i\.get + m_i\.set + C_i\.lc_i)^*\} \)

- \( m_i \): The set of memories associated with the wires
- \( C_i\.lc_i \): The union of all Components’ input control

Not restricted enough
Contents

1 Introduction
2 Related work
3 42 basics
4 Discussions
5 42 Protocols
6 Comments
Controller and components compatibility?

Why?

- A component is activated while it is in a state that doesn’t permit that.
  Example: A component can not deliver output before it computes them!
- A component is activated, but the data that it requires are not available!
  Example: A component can not compute output while no input are Available

How?

- Behavioral description of 42 component (control contracts)
Components’ contracts

CBSE classification of contracts:

- Basic contracts
  syntactic, types, ...

- Behavioral contracts ★
  pre/post conditions, invariants, ...

- Synchronization contracts ★
  protocols

- Quality of service contracts
  power consumption, response time, ...

\[
S_1() \quad S_2() \quad S_1().S_2()^*
\]
42 Protocols

Sequencing constraints
(final states define macro-step)

Data dependencies

Control information
(control output are stored for further use)

Conditional data expressions

42 Protocols

Op Op2

id1 → od1
id2 → od2
Ctl

atomic step
internal memory
42 Protocols

- Sequencing constraints
  (final states define macro-step)

- Data dependencies
- Control information
  (control output are stored for further use)

- Sequencing constraints
  (final states define macro-step)
42 Protocols

(id1 and id2)_{\text{op}} \rightarrow (\text{od1})

Op Op2

(id1 and id2)

(id2)

(id1)

Internal memory

O_{\text{ctl}}

Final states define macro-step

Sequencing constraints

Data dependencies
42 Protocols

\[(id1 \text{ and } id2) \text{ op } / \alpha := \text{ ctl}(od1)\]

- **Sequencing constraints**
  (final states define macro-step)
- **Data dependencies**
- **Control information**
  (control output are stored for further use)
42 Protocols

\[(id1 \text{ and } id2) \text{op} /\alpha := \text{ctl}(od1)\]

\[(id1 \text{ and IF } \alpha \text{ THEN id2}) \text{op2}(IF \neg\alpha \text{ THEN od2})\]

- Sequencing constraints
  (final states define macro-step)
- Data dependencies
- Control information
  (control output are stored for further use)
- Conditional data expressions
Components and controller compatibility:

- Protocols make it easy to verify whether the controller make good usage of components. Verification may either be done locally to a component (making projections) or globally on the whole system.

- Protocols allow either static (which is a model checking problem) or dynamic (a kind of monitoring).

In some cases, using the protocols and the architecture description, one may generate the controller code.
Correctness of the controller code

Controller is:
for X do :
  m_a, m_b, m_c: FIFO(1,int);
  m_d, m_e: FIFO(1,int);
  m_a.put;
  m_a.get;
  A.X;
  C.K;
  m_e.put;
  m_e.get;
  m_b.put;
  m_b.get;
  B.W;
  B.Z;
  m_c.put;
  m_d.put;
  m_d.get;
  C.L;
  A.Y;
}
Correctness of the controller code

Controller is:
for X do:
  m_a, m_b, m_c: FIFO(1, int);
  m_d, m_e: FIFO(1, int);
  m_a.put;
  m_a.get;
  A.X;
  C.K;
  m_e.put;
  m_e.get;
  m_b.put;
  m_b.get;
  B.W;
  B.Z;
  m_c.put;
  m_d.put;
  m_d.get;
  C.L;
  A.Y;
}
Correctness of the controller code

Controller is:
for X do:
\{ 
\begin{align*}
\text{m}_a, \text{m}_b, \text{m}_c & : \text{FIFO}(1, \text{int}); \\
\text{m}_d, \text{m}_e & : \text{FIFO}(1, \text{int}); \\
\text{m}_a & . \text{put} \\
\text{m}_a & . \text{get} \\
\text{A} & . \text{X} \\
\text{C} & . \text{K} \\
\text{m}_e & . \text{put} \\
\text{m}_e & . \text{get} \\
\text{m}_b & . \text{put} \\
\text{m}_b & . \text{get} \\
\text{B} & . \text{W} \\
\text{B} & . \text{Z} \\
\text{m}_c & . \text{put} \\
\text{m}_d & . \text{put} \\
\text{m}_d & . \text{get} \\
\text{C} & . \text{L} \\
\text{A} & . \text{Y} \\
\end{align*}
\}

\begin{tikzpicture}
\node (A) at (0,0) {$A$}; \node (B) at (2,0) {$B$}; \node (C) at (1,-1) {$C$}; \node (0) at (-1,0) {$0$}; \node (1) at (3,0) {$1$}; \node (Y) at (2,-2) {$(\text{Id}_1; \text{Id}_2)X(\text{Od}_1)$}; \node (Y) at (2,-2) {$(\text{Id}_1; \text{Id}_2)X(\text{Od}_1)$}; \node (xx) at (2,-3) {$XX$};
\path[->] (A) edge (B) (B) edge (C) (C) edge (A);
\end{tikzpicture}
Correctness of the controller code

Controller is:
for X do :
    m_a, m_b, m_c: FIFO(1,int);
m_d,m_e: FIFO(1,int);
m_a.put;
m_a.get;
A.X;
C.K;
m_e.put;
m_e.get;
m_b.put;
m_b.get;
B.W;
B.Z;
m_c.put;
m_d.put;
m_d.get;
C.L;
A.Y;
}
Correctness of the controller code

Controller is:
for X do :
    m_a, m_b, m_c: FIFO(1,int);
    m_d, m_e: FIFO(1,int);
    m_a.put;
    m_a.get;
    A.X;
    C.K;
    m_e.put;
    m_e.get;
    m_b.put;
    m_b.get;
    B.W;
    B.Z;
    m_c.put;
    m_d.put;
    m_d.get;
    C.L;
    A.Y;
}
Correctness of the controller code

Controller is:
for X do :
    m_a, m_b, m_c: FIFO(1, int);
    m_d, m_e: FIFO(1, int);
    m_a.put;
    m_a.get;
    A.X;
    C.K;
    m_e.put;
    m_e.get;
    m_b.put;
    m_b.get;
    B.W;
    B.Z;
    m_c.put;
    m_d.put;
    m_d.get;
    C.L;
    A.Y;
}
Correctness of the controller code

Controller is:
for X do :
  m_a, m_b, m_c: FIFO(1,int);
  m_d, m_e: FIFO(1,int);
  m_a.put;
  m_a.get;
  A.X;
  C.K;
  m_e.put;
  m_e.get;
  m_b.put;
  m_b.get;
  B.W;
  B.Z;
  m_c.put;
  m_d.put;
  m_d.get;
  C.L;
  A.Y;
}
Contents

1. Introduction
2. Related work
3. 42 basics
4. Discussions
5. 42 Protocols
6. Comments
Comments and further works

Comments

- 42 is a System level design approach
- Focus on checking assemblage of components
- Component specification is enhanced with protocols
- We allow programming MoCCs using simple primitives

Further works

- More work on protocols
- Assemblage checking, Controller verification
- Applying the approach to more case studies