Co-Simulation

Stavros Tripakis
University of California, Berkeley
Heterogeneous Systems

**Cyber-physical systems** (CPS) are typically **heterogeneous**:

- Cyber = discrete, digital, computer-based systems (hardware & software)
- + Physical = typically continuous systems

Modeling CPS requires **heterogeneous modeling** (or **multi-modeling**) techniques:

- Modeling formalisms, languages, and tools, that combine:
  - Automata, state machines, transition systems, dataflow, discrete event systems, timed automata, ODEs, DAEs, PDEs, hybrid automata, ...
Heterogeneous Systems

Cyber-physical systems (CPS) are typically heterogeneous:

- Cyber = discrete, digital, computer-based systems (hardware & software)
- + Physical = typically continuous systems

Modeling CPS requires heterogeneous modeling (or multi-modeling) techniques:

- Modeling formalisms, languages, and tools, that combine:
  - Automata, state machines, transition systems, dataflow, discrete event systems, timed automata, ODEs, DAEs, PDEs, hybrid automata, ...

This is still a research problem.
Multi-Modeling

Problems: tool interoperability? model exchange? co-simulation?

More than just software engineering: semantic heterogeneity!

Low-level controllers
Simulink

Physical dynamics
Modelica

Supervisory controllers
Rhapsody/SysML
FMI – “Functional Mock-up Interface”

A standard API for model exchange and co-simulation

- Submodels: “FMUs” (“Functional Mock-up Units”)
FMI – “Functional Mock-up Interface”

A standard API for model exchange and co-simulation

- Submodels: “FMUs”
FMI – “Functional Mock-up Interface”

A standard API for model exchange and co-simulation

- **Submodels: “FMUs”**

Promising: backed-up by industry (Daimler, Dassault Systemes, IBM, …).

Promising: backed-up by industry (Daimler, Dassault Systemes, IBM, …).

Open problem: how to define a “good” master algorithm? What does “good” even mean?
Interesting (open for the most part) questions

- What is the right API?
- Can the API support many modeling languages (discrete, continuous, ...)?
- Are some APIs better than others? In what respects?
- What are the right algorithms for given APIs?
Interesting (open for the most part) questions

- What is the right API?
- Can the API support many modeling languages (discrete, continuous, ...)?
- Are some APIs better than others? In what respects?
- What are the right algorithms for given APIs?

We will assume the FMI API given. We will focus on the master algorithm.
Modularity

We will also in the process solve the problem of **modular** (discrete-event) simulation.

Modular = the simulation algorithm interacts with each actor using standard API.
Modularity

We will also in the process solve the problem of **modular** (discrete-event) simulation.

Modular = the simulation algorithm interacts with each actor using standard API.

Why is this desirable? (even without multi-modeling)
FMI in a Nutshell

[Broman et al. (2013)]

Notation:

- $C$: set of FMU instances in a model
- $c \in C$: FMU instance
- $S_c$: set of states of FMU $c$
- $U_c$: set of input ports of $c$
- $Y_c$: set of output ports of $c$
- $\mathbb{V}$: set of values that a port can take

API’s main functions:

- $\text{init}_c : \mathbb{R}_{\geq 0} \rightarrow S_c$
- $\text{set}_c : S_c \times U_c \times \mathbb{V} \rightarrow S_c$
- $\text{get}_c : S_c \times Y_c \rightarrow \mathbb{V}$
- $\text{doStep}_c : S_c \times \mathbb{R}_{\geq 0} \rightarrow S_c \times \mathbb{R}_{\geq 0}$
FMI in a Nutshell

[Broman et al. (2013)]

Notation:

- $C$: set of FMU instances in a model
- $c \in C$: FMU instance
- $S_c$: set of states of FMU $c$
- $U_c$: set of input ports of $c$
- $Y_c$: set of output ports of $c$
- $\mathbb{V}$: set of values that a port can take

API's main functions:

- $\text{init}_c: \mathbb{R}_{\geq 0} \rightarrow S_c$
- $\text{set}_c: S_c \times U_c \times \mathbb{V} \rightarrow S_c$
- $\text{get}_c: S_c \times Y_c \rightarrow \mathbb{V}$
- $\text{doStep}_c: S_c \times \mathbb{R}_{\geq 0} \rightarrow S_c \times \mathbb{R}_{\geq 0}$

$\text{init}_c(t) \mapsto s$
FMI in a Nutshell [Broman et al. (2013)]

Notation:

\[ C \] set of FMU instances in a model
\[ c \in C \] FMU instance
\[ S_c \] set of states of FMU \( c \)
\[ U_c \] set of input ports of \( c \)
\[ Y_c \] set of output ports of \( c \)
\[ \mathbb{V} \] set of values that a port can take

API’s main functions:

\[ \text{init}_c : \mathbb{R}_{\geq 0} \rightarrow S_c \]
\[ \text{set}_c : S_c \times U_c \times \mathbb{V} \rightarrow S_c \]
\[ \text{get}_c : S_c \times Y_c \rightarrow \mathbb{V} \]
\[ \text{doStep}_c : S_c \times \mathbb{R}_{\geq 0} \rightarrow S_c \times \mathbb{R}_{\geq 0} \]
FMI in a Nutshell

[Broman et al. (2013)]

Notation:

\[ C \] \text{ set of FMU instances in a model}
\[ c \in C \] \text{ FMU instance}
\[ S_c \] \text{ set of states of FMU } c
\[ U_c \] \text{ set of input ports of } c
\[ Y_c \] \text{ set of output ports of } c
\[ V \] \text{ set of values that a port can take}

API’s main functions:

\[
\begin{align*}
\text{init}_c : & \mathbb{R}_{\geq 0} \to S_c \\
\text{set}_c : & S_c \times U_c \times V \to S_c \\
\text{get}_c : & S_c \times Y_c \to V \\
\text{doStep}_c : & S_c \times \mathbb{R}_{\geq 0} \to S_c \times \mathbb{R}_{\geq 0}
\end{align*}
\]

\[ \text{get}_c(s, y) \mapsto v \]
Notation:

\[ C \] set of FMU instances in a model  
\[ c \in C \] FMU instance  
\[ S_c \] set of states of FMU \( c \)  
\[ U_c \] set of input ports of \( c \)  
\[ Y_c \] set of output ports of \( c \)  
\[ \mathbb{V} \] set of values that a port can take

API’s main functions:

\[ \text{init}_c : \mathbb{R}_{\geq 0} \rightarrow S_c \]  
\[ \text{set}_c : S_c \times U_c \times \mathbb{V} \rightarrow S_c \]  
\[ \text{get}_c : S_c \times Y_c \rightarrow \mathbb{V} \]  
\[ \text{doStep}_c : S_c \times \mathbb{R}_{\geq 0} \rightarrow S_c \times \mathbb{R}_{\geq 0} \quad \text{doStep}_c(s, h) \mapsto (s', h') \]
Simulink blocks
(from Simulink User’s Guide – R2013b)

A Simulink block consists of inputs, a set of states, and outputs:

\[ \begin{align*}
  & u \\
  \text{(Input)} & x \\
  \text{(States)} & y \\
  \text{(Output)}
\end{align*} \]

Basic block methods:

- **Outputs**: Computes the outputs of a block given its inputs at the current time step and its states at the previous time step.

- **Update**: Computes the value of the block’s discrete states at the current time step, given its inputs at the current time step and its discrete states at the previous time step.

- **Derivatives**: Computes the derivatives of the block’s continuous states at the current time step, given the block’s inputs and the values of the states at the previous time step.
Every actor in Ptolemy also implements a standard API:

initialize(), prefire(), fire(), postfire(), ...

Formalized in [Tripakis et al. (2013)].
The Problem of Determinacy

FMUs may be connected in arbitrary ways (including in feedback):

In what order should the master algorithm execute such a model?

How to guarantee that simulation results do not depend on arbitrary factors (e.g., FMU name, position, creation time, ...)?
Dealing with feedback

Using I/O dependency information (c.f. lecture on synchronous systems).

FMI provides an (unfortunately optional) mechanism for an FMU to declare I/O dependencies:
Dealing with feedback

If no cyclic dependencies, then model can be executed deterministically:

- get known outputs $\rightarrow$ set dependent inputs $\rightarrow$ repeat while respecting the dependencies, until all I/O ports are set;
- update states of all FMUs by calling `doSteps` (we’ll see how).
Dealing with feedback

If no cyclic dependencies, then model can be executed deterministically:

- get known outputs $\rightarrow$ set dependent inputs $\rightarrow$ repeat while respecting the dependencies, until all I/O ports are set;
- update states of all FMUs by calling doSteps (we’ll see how).

If cyclic dependencies, model is rejected.
Dealing with feedback

Could we have used fixpoint methods (constructive semantics) instead?

Could we have used fixpoint methods (constructive semantics) instead?
Dealing with feedback

Could we have used fixpoint methods (constructive semantics) instead?

No: FMUs are black-boxes ⇒ no way to guarantee statically (at compile time) that model is constructive.
Next Challenge: Updating FMU States

Consider the example:

In what order should \( \text{doStep}_{\text{FMU1}} \) and \( \text{doStep}_{\text{FMU2}} \) be called?

Suppose:

\[ \text{doStep}_{\text{FMU1}}(h); \text{doStep}_{\text{FMU2}}(h); \]

▶ What if FMU1 accepts \( h \) but FMU2 rejects it?

Suppose:

\[ \text{doStep}_{\text{FMU2}}(h); \text{doStep}_{\text{FMU1}}(h); \]

▶ What if FMU2 accepts \( h \) but FMU1 rejects it?

⇒ We need rollback!

▶ Saving the state of an FMU, to restore it and try another (smaller) time step \( h \).

▶ Rollback available (although optional) in draft FMI 2.0.
Next Challenge: Updating FMU States

Consider the example:

In what order should \texttt{doStep}_{\text{FMU1}} and \texttt{doStep}_{\text{FMU2}} be called?

- Suppose: \texttt{doStep}_{\text{FMU1}}(h); \texttt{doStep}_{\text{FMU2}}(h);
Next Challenge: Updating FMU States

Consider the example:

In what order should \( \text{doStep}_{\text{FMU1}} \) and \( \text{doStep}_{\text{FMU2}} \) be called?

- Suppose: \( \text{doStep}_{\text{FMU1}}(h); \text{doStep}_{\text{FMU2}}(h); \)
  - What if FMU1 \textit{accepts} \( h \) but FMU2 \textit{rejects} it?

⇒ We need rollback!

- Saving the state of an FMU, to restore it and try another (smaller) time step \( h \).

- Rollback available (although optional) in draft FMI 2.0.
Next Challenge: Updating FMU States

Consider the example:

In what order should doStep\textsubscript{FMU1} and doStep\textsubscript{FMU2} be called?

- Suppose: doStep\textsubscript{FMU1}(h); doStep\textsubscript{FMU2}(h);
  - What if FMU1 accepts $h$ but FMU2 rejects it?

- Suppose: doStep\textsubscript{FMU2}(h); doStep\textsubscript{FMU1}(h);
  - What if FMU2 accepts $h$ but FMU1 rejects it?
Next Challenge: Updating FMU States

Consider the example:

In what order should $\text{doStep}_{\text{FMU1}}$ and $\text{doStep}_{\text{FMU2}}$ be called?

- Suppose: $\text{doStep}_{\text{FMU1}}(h); \text{doStep}_{\text{FMU2}}(h)$;
  - What if FMU1 accepts $h$ but FMU2 rejects it?

- Suppose: $\text{doStep}_{\text{FMU2}}(h); \text{doStep}_{\text{FMU1}}(h)$;
  - What if FMU2 accepts $h$ but FMU1 rejects it?

⇒ We need rollback!
  - Saving the state of an FMU, to restore it and try another (smaller) time step $h$. 
Next Challenge: Updating FMU States

Consider the example:

In what order should $\text{doStep}_{\text{FMU1}}$ and $\text{doStep}_{\text{FMU2}}$ be called?

- Suppose: $\text{doStep}_{\text{FMU1}}(h); \text{doStep}_{\text{FMU2}}(h)$;
  - What if FMU1 accepts $h$ but FMU2 rejects it?

- Suppose: $\text{doStep}_{\text{FMU2}}(h); \text{doStep}_{\text{FMU1}}(h)$;
  - What if FMU2 accepts $h$ but FMU1 rejects it?

$\Rightarrow$ We need rollback!
  - Saving the state of an FMU, to restore it and try another (smaller) time step $h$.
  - Rollback available (although optional) in draft FMI 2.0.
Master Algorithm A – (At Most) Two Passes with Rollback

At every simulation step:

1. set values for all input/output ports (using I/O dependencies)
2. save the states of all FMUs (to enable rollback)
3. set communication step size to an initial default value:
   \[ h := h_{\text{max}} \]
4. find \( h \) acceptable by all FMUs:
   for each \( c \in C \) do
   1. \( h' := \text{doStep}_c(h_{\text{max}}) \)
   2. \( h := \min(h, h') \)
5. assert \( 0 \leq h \leq h_{\text{max}} \)
6. if \( h < h_{\text{max}} \) then \hspace{1em} // roll back
   restore saved states of all FMUs;
   for each \( c \in C \) do
   1. \( h' := \text{doStep}_c(h) \)
   2. assert \( h' = h \)
7. return \( h \).
Master Algorithm A – (At Most) Two Passes with Rollback

At every simulation step:

1. set values for all input/output ports (using I/O dependencies)
2. save the states of all FMUs (to enable rollback)
3. set communication step size to an initial default value:
   \[ h := h_{\text{max}} \]
4. find \( h \) acceptable by all FMUs:
   for each \( c \in C \) do
     1. \( h' := \text{doStep}_c(h_{\text{max}}) \)
     2. \( h := \min(h, h') \)
5. assert \( 0 \leq h \leq h_{\text{max}} \) does it hold? why?
6. if \( h < h_{\text{max}} \) then \( \text{// roll back} \)
   restore saved states of all FMUs;
   for each \( c \in C \) do
     1. \( h' := \text{doStep}_c(h) \)
     2. assert \( h' = h \) does it hold? why?
7. return \( h \).
(A0) If \( \text{doStep}_c(s, h) = (s', h') \) then \( 0 \leq h' \leq h \).

If \( h' = h \) then FMU accepts \( h \), otherwise it rejects it.
FMU Contract

(A0) If $\text{doStep}_c(s, h) = (s', h')$ then $0 \leq h' \leq h$.  
If $h' = h$ then FMU accepts $h$, otherwise it rejects it.

(A1) If $\text{doStep}_c(s, h) = (s', h')$, then for any $h''$ where $0 \leq h'' \leq h'$,  
$\text{doStep}_c(s, h'') = (s'', h'')$ for some $s''$.  
If FMU accepts $h'$, then it accepts any $h'' < h'$. 

**FMU Contract**

(A0) If $\text{doStep}_c(s, h) = (s', h')$ then $0 \leq h' \leq h$.

If $h' = h$ then FMU accepts $h$, otherwise it rejects it.

(A1) If $\text{doStep}_c(s, h) = (s', h')$, then for any $h''$ where $0 \leq h'' \leq h'$, $\text{doStep}_c(s, h'') = (s'', h'')$ for some $s''$.

If FMU accepts $h'$, then it accepts any $h'' < h'$.

(A2) Let $s' = \text{set}_c(s, u, v)$. Then $s' = s[u := v]$.

set only changes the value of the variable being set.
FMU Contract

(A0) If $\text{doStep}_c(s, h) = (s', h')$ then $0 \leq h' \leq h$.
If $h' = h$ then FMU accepts $h$, otherwise it rejects it.

(A1) If $\text{doStep}_c(s, h) = (s', h')$, then for any $h''$ where $0 \leq h'' \leq h'$,
$\text{doStep}_c(s, h'') = (s'', h'')$ for some $s''$.
If FMU accepts $h'$, then it accepts any $h'' < h'$.

(A2) Let $s' = \text{set}_c(s, u, v)$. Then $s' = s[u := v]$.
set only changes the value of the variable being set.

(A3) Let $v = \text{get}_c(s, y)$ and $v' = \text{get}_c(s', y)$. If
$s' = s[u_1 := v_1, \ldots, u_k := v_k]$, and output $y$ does not directly depend
on any input $u_1, \ldots, u_k$, then $v' = v$.
the value returned by get is the same when only irrelevant inputs change.
Properties of Master Algorithm A

- **Termination**: at most two iterations over all FMUs.

- **Determinacy**: at the end of every simulation step, the value of the FMU states and the chosen step $h$ are independent of the order in which ports and FMUs are chosen by the algorithm.

- **Correctness**: at the end of every simulation step, the value of the FMU states is the same as if the correct step size $h$ had been guessed and $\text{doStep}$ called only once on each FMU.

- **Maximal progress**: if the algorithm manages to advance by $h < h_{\text{max}}$, then there is no $h' > h$ which is acceptable by all FMUs, starting from the same state. This requires an additional assumption:
  
  (P1) If $\text{doStep}(s, h) = (s', h')$ and $h' < h$, then for all $h'' > h'$, $\text{doStep}(s, h'') = (s'', h')$ for some $s''$. 

Stavros Tripakis (UC Berkeley)
EE 144/244, Fall 2015
Properties of Master Algorithm A

- **Termination**: at most two iterations over all FMUs.

- **Determinacy**: at the end of every simulation step, the value of the FMU states and the chosen step $h$ are independent of the order in which ports and FMUs are chosen by the algorithm.
Properties of Master Algorithm A

- **Termination**: at most two iterations over all FMUs.

- **Determinacy**: at the end of every simulation step, the value of the FMU states and the chosen step $h$ are independent of the order in which ports and FMUs are chosen by the algorithm.

- **Correctness**: at the end of every simulation step, the value of the FMU states is the same as if the correct step size $h$ had been guessed and doStep called only once on each FMU.
Properties of Master Algorithm A

- **Termination**: at most two iterations over all FMUs.

- **Determinacy**: at the end of every simulation step, the value of the FMU states and the chosen step $h$ are independent of the order in which ports and FMUs are chosen by the algorithm.

- **Correctness**: at the end of every simulation step, the value of the FMU states is the same as if the correct step size $h$ had been guessed and $\text{doStep}$ called only once on each FMU.

- **Maximal progress**: if the algorithm manages to advance by $h < h_{\text{max}}$, then there is no $h' > h$ which is acceptable by all FMUs, starting from the same state.

  This requires an additional assumption:

  \((P1)\) If $\text{doStep}_c(s, h) = (s', h')$ and $h' < h$, then for all $h'' > h'$, $\text{doStep}_c(s, h'') = (s'', h')$ for some $s''$. 

Stavros Tripakis (UC Berkeley)
Limitations of Master Algorithm A

- *Legacy* FMUs: what if an FMU does not support rollback?
  - Case not uncommon, e.g., old FORTRAN codes at LBNL.
Limitations of Master Algorithm A

- *Legacy* FMUs: what if an FMU does not support rollback?
  - Case not uncommon, e.g., old FORTRAN codes at LBNL.

- Even for those FMUs supporting rollback, can it be avoided?
  - Saving and restoring large states can be expensive.
  - Note, however, that our algorithm requires only 1-step rollback.
Extensions of Master Algorithm A

- Can extend MA A to handle at most 1 (one ...) legacy FMU.

  1. Find maximal $h$ acceptable by all non-legacy FMUs.
  2. Call $h' := \text{doStep}(h)$ on the legacy FMU.
  3. If $h' < h$, rollback and call $\text{doStep}(h')$ on all non-legacy FMUs.

Can avoid rollback if API is extended with an additional method that allows to query an FMU about its maximal acceptable step size.
Extensions of Master Algorithm A

- Can extend MA A to handle at most 1 (one ...) legacy FMU. How?
  1. Find maximal $h$ acceptable by all non-legacy FMUs.
  2. Call $h' \equiv \text{doStep}(h)$ on the legacy FMU.
  3. If $h' < h$, rollback and call $\text{doStep}(h')$ on all non-legacy FMUs.
Extensions of Master Algorithm A

- Can extend MA A to handle at most 1 (one ...) legacy FMU. How?
  1. Find maximal $h$ acceptable by all non-legacy FMUs.
  2. Call $h' := \text{doStep}(h)$ on the legacy FMU.
  3. If $h' < h$, rollback and call $\text{doStep}(h')$ on all non-legacy FMUs.

- Can avoid rollback if API is extended with an additional method that allows to query an FMU about its maximal acceptable step size.
Querying FMUs for Maximal Step Sizes

Additional API method:

\[
\text{getMaxStepSize}_c : S_c \to \mathbb{R}_{\geq 0} \cup \{\infty\}
\]

Contract:

(A4) If \( \text{getMaxStepSize}_c(s) = h \) then for all \( h' \) where \( 0 \leq h' \leq h \), \( \text{doStep}_c(s, h') = (s', h') \) for some \( s' \).
Master Algorithm B (sketch)

At every simulation step:

1. set values for all input ports (using I/O dependencies, as in MA A)
2. find the maximal step size accepted by all FMUs that can be queried (this set is denoted $C_P$):
   \[ h := \min\left(\{\text{getMaxStepSize}_c() \mid c \in C_P\} \cup \{h_{\text{max}}\}\right) \]
3. continue knowing that this is the maximal $h$ that can be achieved, and the guarantee that all $c \in C_P$ will accept anything smaller:
   
   1. call $\text{doStep}(h)$ on those FMUs that allow rollback (all except possibly those in $C_P$ and the at most 1 legacy FMU)
   2. find the minimum $h'$ that they accept: this must be $h' \leq h$
   3. call $\text{doStep}(h')$ on the legacy FMU: it returns $h'' \leq h'$
   4. call $\text{doStep}(h'')$ on all FMUs except legacy (possibly using rollback on FMUs not in $C_P$).

See [Broman et al. (2013)] for details.
Master Algorithm B (sketch)

At every simulation step:

1. set values for all input ports (using I/O dependencies, as in MA A)
2. find the maximal step size accepted by all FMUs that can be queried (this set is denoted $C_P$):
   \[ h := \min\left(\{\text{getMaxStepSize}_c() \mid c \in C_P\} \cup \{h_{max}\}\right) \]
3. continue knowing that this is the maximal $h$ that can be achieved, and the guarantee that all $c \in C_P$ will accept anything smaller:
   1. call `doStep(h)` on those FMUs that allow rollback (all except possibly those in $C_P$ and the at most 1 legacy FMU)
   2. find the minimum $h'$ that they accept: this must be $h' \leq h$
   3. call `doStep(h')` on the legacy FMU: it returns $h'' \leq h'$
   4. call `doStep(h'')` on all FMUs except legacy (possibly using rollback on FMUs not in $C_P$).

See [Broman et al. (2013)] for details.
T. Blochwitz, M. Otter, and et al.

In 6th International Workshop on Multi-Paradigm Modeling (MPM’12), 2012.

Determinate Composition of FMUs for Co-Simulation.

MODELISAR Consortium and Modelica Association.

Taming heterogeneity – the Ptolemy approach.

A modular formal semantics for Ptolemy.