Sim-Diasca Coupling HOWTO

Organisation: Copyright (C) 2016-2022 EDF R&D

Contact: olivier (dot) boudeville (at) edf (dot) fr

Author: Olivier Boudeville

Creation Date: Tuesday, January 3, 2017

Lastly updated: Monday, October 17, 2022

Version: 2.4.4

Status: Stable

Website: http://sim-diasca.com

Dedication: For the Sim-Diasca model integrators.

Abstract: This document describes how the *coupling* of models and simulations can be done with Sim-Diasca.

Four coupling schemes are more specifically discussed, following:

- a black box model
- a model-driven adapter
- an adapter-driven model
- the native model (the reference, usual way models are integrated)

These coupling schemes account for the most common options that range from co-simulation to pure simulation.

Table of Contents

1 Understanding the Coupling Use Case

1.1 Purpose of Coupling

When having to perform simulations, the usual design process involves:

1. specifying the questions that shall be answered by the study
2. establishing the corresponding metrics that shall be evaluated by the simulation
3. describing the target system of interest and its context, by specifying the models and scenarios involved
4. selecting the simulation engine of choice for that case
5. implementing the various simulation elements accordingly

However in some cases the last two steps cannot be performed in that order, notably because some models may predate the simulation effort at hand.

In this case we have to deal with pre-existing models, and the challenge switches from their writing to their coupling, so that, **even if these models are already written, they can be nevertheless integrated - at least to some extent - in the new simulation of interest**.

Said otherwise, the models involved in the vast majority of Sim-Diasca simulations shall be purposedly Erlang-based and shall derive, directly or not, from the `class_Actor` mother classes provided by the engine.

We suppose, in the context of this document, that this most direct "native" approach cannot be taken, inducing the need to perform some kind of coupling instead.

1.2 Coupling Among the Sim-Diasca Example Cases

Sim-Diasca primarily focuses on the usual simulation design process described above, where the engine is first selected, and then the models are written for that engine.

This typically corresponds to the *City-Example* simulation case[1], which focuses on urban simulation by itself (i.e. on the models themselves).

Another example case, the *Sustainable-Cities*[2] one, is also a fairly classical simulation, yet focuses on its boundaries (i.e. its input and output) in the context of the integration to a third-party platform; yet no actual coupling is involved here either.

A third simulation case, nicknamed *MUG*[3], focuses on the subject of interest here, i.e. the **coupling of a set of models that were not all written for Sim-Diasca** (they were written independently, and their coupling is thus an afterthought).

[1] This case is distributed with the free software version of Sim-Diasca, and can be found in the `mock-simulators/city-example` directory.

[2] This case does not belong to the free software version of Sim-Diasca; it is located in the `sustainable-cities/sustainable-cities-case` directory.

[3] This case does not belong to the free software version of Sim-Diasca (it is located in the `sustainable-cities/mug-case` directory of the internal version).

As a result these models were not specifically in line with the engine's expectations at various levels; from the most problematic to the least:

- **models had heterogeneous semantics**: their modelling differed from one to another, and also altogether from the engine's ones (ex: some were based on a dataflow logic[4] rather than on the usual multi-agent, disaggregated approach upon which Sim-Diasca relies; their view on the target system, on space, on time also differed, i.e. they did not obey a common ontology)

- regarding simulation, **models had their own mode of operation**, organization, and conventions regarding their respective creation, state management, scheduling, evaluation, interactions, result generation, etc.

- regarding their software architecture, **models relied on different build-time and runtime environments**: operating systems could differ, programming language were also heterogeneous (ex: some models had been developed in Python, others in Java, inducing the need for the engine to be able to deal simultaneously with multiple programming languages that had to cooperate)

Additionally some models were delivered "as they were", as black boxes (hence with no information at all about their inner workings and no possibility of being modified either).

The challenge was therefore to devise a sufficiently generic interoperability scheme enabling the coupling of models from any origin, ironing out their multi-level differences while minimizing the dependency of models onto others, or onto the engine itself ; the coupling architecture discussed here is an answer to these needs.

[4] Please refer to the *Sim-Diasca Dataflow HOWTO* for more information.

2 Coupling Architecture

2.1 Simulations Including Native Models and Adapted Ones

Whether or not a case relies on coupling, the overall simulation is to be driven by the engine, notably by its time management service. As a consequence, from the engine's point of view, all the scheduled elements are expected to be ultimately (Sim-Diasca) actors.

For some models, either not developed yet or for which having an alternate implementation is relevant, the best option for the project may be to have them implemented directly in the target simulation environment. Then by design they will be well integrated with the engine.

However, probably more often than not, such a complete integration will not be performed, for various reasons: a model may have been delivered as a black box only, or another form of implementation is preferred, or any kind of port for this model would require too much rewriting work.

So we need in these cases to also include models that have not been designed according to any prior federating scheme, implying that no specific coupling consideration or engine integration was taken into account when they were first designed. We have therefore to provide technical measures ensuring that these models will nevertheless fit, a posteriori, in the overall Sim-Diasca scheme.

To perform these adaptations, we will resort to a range of **coupling schemes** that will make these pre-existing models behave, from the point of view of the engine, as if they were legit Sim-Diasca models.

Technically, this means that engine-specific adapters, implemented in Erlang, will wrap each exogenous model and, far beyond the question of the programming languages, will ensure that the resulting overall model complies with the appropriate synchronization and exchange contracts, so that that model can be seamlessly integrated in the simulation among the other models, regardless of their nature.

Such a simulation, where the models accommodate the engine, can alternatively be seen as a *co-simulation*, as discussed in the next section.

2.2 A Co-Simulation

Let's start with an excerpt of the Wikipedia article about co-simulation:

In co-simulation the different subsystems which form a coupled problem are modeled and simulated in a distributed manner. Hence, the modeling is done on the subsystem level without having the coupled problem in mind. Furthermore, the coupled simulation is carried out by running the subsystems in a black-box manner. During the simulation the subsystems will exchange data.

In this context, the architecture of the coupling case discussed here can also be seen as Sim-Diasca being the *master*[5] engine, with each of the third-party *slave* models corresponding to a coupled simulation (not to an individual model anymore), a simulation evaluated by any engine that this slave embeds.

[5] A term used also in FMI parlance, for *Co-Simulation*, as opposed to a *Model-Exchange* integration.

Mixing and matching models that are either natively developed for the engine or that are merely adapted for it somewhat blurs the frontier between a simulation engine and a coupling engine (i.e. a master).

Should there be *only* coupled models (as opposed to native ones), we could rely on a pure co-simulation master, an example of which being MECSYCO; if, additionally, all coupled models were FMUs, DACCOSIM could also be used.

We will see in the next sections that, for the coupling cases considered here, more diverse schemes than mere black boxes are to be managed.

2.3 The Four Main Coupling Schemes Considered Here

So we saw that there is a kind of continuum between "pure *co-simulations*" (i.e. with only coupled simulations being involved) and "pure *simulations*" (i.e. with only engine-native models involved).

More precisely, for any given model, four main **coupling schemes** (noted `CS`) can be seen as pacing this continuum; going from the least integrated to the engine to the most, one may rely on:

- **Adapted black boxes** [`CS1`]: standalone simulations can be integrated to an overall simulation exactly as they are (as untouched executables) thanks to specific two-way translators that bridge the gap with the engine, upstream and downstream of a black-box that remains fully oblivious of the rest of the simulation (this corresponds to a general approach used when performing ad hoc co-simulation)
- **Autonomous adapted models** [`CS2`]: these models may still drive their evaluation by themselves as they used to, yet have to be modified in order to rely on some services (API) provided by the engine (through their adapter) so that they can interact with the rest of the simulation
- **Adapter-driven models** [`CS3`]: the flow of control of these models is driven here by their adapter, which feeds them appropriately with the data determined by the engine (as if these models were mostly domain-specific libraries; the approaches used when performing FMI-like co-simulations are special cases of it)
- **Native models** [`CS4`]: these models have been specifically implemented for the engine being used, thus do not need any kind of adaptation and provide best performances (this is the general case when performing a *simulation*)

Each of these coupling schemes will be detailed below.

Of course none of these four modes of operation strictly prevail over any other, they all have pros and cons, so any choice in this matter is the consequence of an architectural trade-off depending heavily on the model, the engine and the objectives of the project.

Besides the coupling potential (i.e. the possibility of including that model in numerous interactions, so that the added value expected from the coupling can be fully obtained), other metrics may be of interest in order to select a scheme:

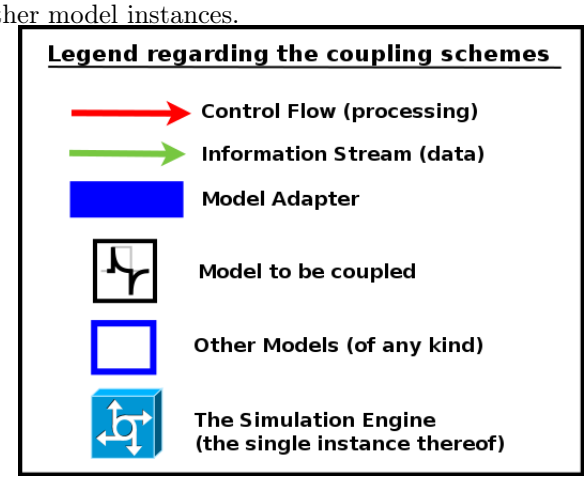
- the respective efforts that would be needed to integrate that model (ex: development of a pair of translators in the case of black boxes, adaptation or partial rewriting of this model, etc.)

- whether it leads to a fork, a renewal of that model, possibly providing a new version of reference, and whether it is welcome
- the architecture and performances induced by the integration (ex: in most cases, black boxes are stateless, they are executed from scratch at each timestep as a new process, with much file-based I/O involved, while other approaches may rely on a stateful process running permanently on par with the engine and interacting with it, or on a higher-level component directly operated by the engine - each time with different runtime overheads)

> **Note**
> A key point is that this choice of coupling scheme is *model-level*, not simulation-level: a given simulation may involve various models, each relying on a coupling scheme that best suits its nature and needs.

In the next section, the various coupling schemes are detailed, so that one can select the best approach when having to integrate a model in one's simulation.

These descriptions focus on the coupled model itself, knowing that its environment ("the rest of the simulation") is mostly made of the engine and of the other model instances.



The red arrows denote the *control flow,* i.e. which component drives which one, in what order.

The green arrows denote the *information stream*, i.e. the paths taken by the data conveying that information, from a component to another.

As in most cases a pre-existing model cannot be integrated "as is" in the simulation, an *adapter* (shown as a blue, filled rectangle), is generally required. From the engine's point of view, such an adapter is an actor like the others, and is treated as such; the engine then is unaware that there is a third-party model underneath.

This adapter applies to an instance of the *model* considered here (depicted in these diagrams as the plot of a curve), which usually is meant to interact with *other model instances* (be them of the same model or not), represented by white rectangles enclosed in blue.

Finally, the overall simulation is driven by the *simulation engine* (namely Sim-Diasca, shown here as a light-blue component with arrows suggesting mul-

tiple paths), in charge of the coordination of all model instances (actors) and of their exchanges.

Let's then discuss each coupling scheme in turn.

2.3.1 Adapted Black Box [CS1]

In this setting, we have **a pre-existing model**, possibly a raw binary **standalone executable**, on the mode of operation of which we have little to no information, except:

- the file formats of its inputs
- the file formats of its outputs
- how it should be executed (ex: command-line options, environment variables, dependencies, etc.)

So this model is originally expected to be run (in an appropriate environment) with a command-line akin to:

```
$ ./my-simulator.exe --option_1 --option_2=42
     --first-input-file=foo.dat --first-output-file=bar.dat
```

In this scheme we then create a corresponding black-box adapter, in the form of a Sim-Diasca actor, comprising mostly:

- its optional, internal, private, durable adapter-level **model state** (as any actor); this corresponds to a durable state relevant for the embedded black box, should it need one
- an **input translator**, transforming information from the rest of the simulation and from its state to a set of files respecting the input formats supported by the model executable
- an **output translator**, transforming information found in the files being output by the black box into, possibly, information used to update the simulation (i.e. the state of this actor and the one of other relevant actors) and the results

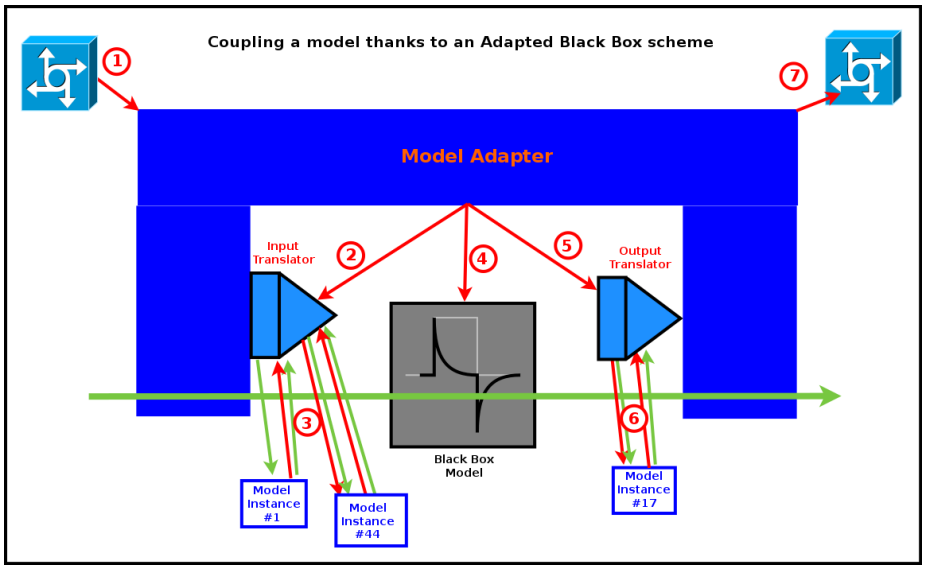
Both translators are typically direct parts of the adapter, and may use helper libraries to perform their conversion of information format, especially if a common data model has been defined underneath.

Variations of this scheme exist; notably:

- either the underlying executable is run each time it is scheduled, exchanging information with the simulation thanks to actual files
- or that executable is run once and remains running on par of the simulation, communicating typically over file descriptors

In all cases we consider that it is the responsibility of the adapter to collect and propagate on behalf of the black box the relevant information from and to the simulation; for that, the adapter has to behave like a standard (Sim-Diasca) actor (using actor messages for that, respecting the scheduling rules, being stateful, etc.).

In any case, as shown in the diagram below, an evaluation of the model results then in the following sequence of actions:



1. the **model adapter**, being registered for a scheduling (more on that later) is triggered by the engine and thus executes its behaviour, possibly starting with some pre-processing and updating of its state

2. its **input translator** fetches then in the simulation any relevant information (possibly complemented by the adapter's state) and performs any relevant processing of it, as any other actor, so that appropriate input files are produced for the black box

3. for that, **two-way information retrieval and processing** is to be performed by the translator, using standard, engine-mediated actor messages to communicate with the other model instances

4. the relevant input files having been produced, the adapter **executes** then the **black box model**, as an external program, specifying the relevant information for that (notably what are the input files and any other command-line options); the model is not expected to perform any interaction with the simulation (and anyway would be most probably unable to do so)

5. once this processing is over, the **output translator**, also included in the adapter, reads and parses the output files generated by the model, and then updates accordingly the simulation and, possibly, the relevant probes (still respecting the engine's conventions)

6. this involves again engine-synchronised **two-way exchanges of actor messages** (exactly like for the input translator)

7. the **adapter** may finally perform any relevant post-processing, and then notifies the engine that it finished evaluating its current scheduling

Of course multiple of these sequences of actions (one per model instance) may run concurrently during the evaluation of a time-step.

Usually this implies that one operation-system process (typically, a UNIX process) has to be spawned at each time-step (for the black box executable), and that at least two files are to be written and then immediately read, parsed and discarded (to account for the communication between the black box and its two translators).

Relying on a black box is probably the **most basic form of coupling**; it remains often quite loose, yet many projects may find this approach sufficient.

Pros:

- the model does not depend in any way on the engine or on the coupling architecture: it does not have to be modified (hence no fork) and even its inner workings may not be known at all (yet, assessing that the coupling makes sense for the domain of interest is made more problematic)
- each part (the model and the adapter, with its input and output translators) can be developed and tested fairly separately

Cons:

- this is low-level, file-based communication, very often model-specific, with limited control of the errors cases and usually no traces collected (otherwise another translator is needed, and additional conventions should preferably be enforced)
- short of having defined and applied a common interchange format, an ad hoc, possibly complex, if not limiting, pair of input and output translators must be developed, which may be expensive (especially in the absence of a common information model)
- generally not very efficient, as a process of the operating system must be spawned at each of the evaluations of this model, and two-way translations are to be performed so that the model can recreate its memory from scratch at each scheduling
- loose coupling only, no finer interaction can be devised short of being able to open this black box (which is isolated by design)
- the implementation of the black box being opaque, either it must be trusted or it may have to be studied; sandboxing solutions could be used there to alleviate any risk in that matter

Some co-simulation approaches make heavy use of this black-box coupling scheme; among the lessons learned, following points were identified:

- a right granularity must be found when breaking down the target system into a set of submodels (neither too fine nor too coarse); the perimeter of each black box and the number of their instances must be finely determined
- the closer models are to continuous time, the more difficult their initialisation may be in order to ensure that all black boxes start with common, correct, consistent initial states
- stable, reliable adapters written in higher-level languages (*wrappers*) are key for efficient couplings (both in terms of design and runtime performances)

- black boxes must have been implemented with scalability in mind, as their memory footprint (ex: because of some solvers) might become very large
- well describing and documenting the black boxes is crucial to their integration, and often overlooked

2.3.2 Autonomous Adapted Model [CS2]

In this coupling scheme, one starts with **a pre-existing model taken as it is, yet its inner logic used to fetch its inputs and transmit its outputs is replaced by calls to a coupling API**, in order to perform the same operations, yet this time through the engine.

This scheme, while looking interesting on the paper, conceals pitfalls of its own, as **it reverses the intended control flow between the engine (meant to drive) and the models (meant to be driven)**.

Indeed, once triggered, each model instance may, thanks the coupling APIs at its disposal, act quite freely upon the simulation, i.e. mostly by itself and with little to no control of the engine - whereas it is up to the latter to organise, synchronise and control accesses of the numerous actors involved.

Typically, such a scheme would naturally lead to the adapted models needing to freely perform indiscriminate, blocking read/write operations onto the rest of the simulation (notably onto other model instances), in spite of a technical context that relies on asynchronous, synchronised message passing to provide parallelism.

Granting such a freedom to models would be done at the expense of the coupling APIs, which would be either very limiting or, more probably, extremely tricky to develop: these APIs would have to hide the fact that underneath each of their call from the model, the engine would have to make the overall logical time seamlessly progress and perform simulation-level synchronisation.

Another option would be to **reimplement in the target language the whole applicative protocol that rules a Sim-Diasca actor**, as it is. This would result in, semantically, offering the same API, yet in a translated form (ex: in Python or Java).

Doing so would involve significant effort on the side of the coupling architecture in order to support each new language, and writing the corresponding models would not be easier than using the classical, native route.

In a parallel context, with an arbitrary number of model instances performing intricate, simultaneous interactions, having the engine being simultaneously driven by each of these actors would be considerably more complex than the default, opposite mode of operation.

However, if emulating a full-blown Sim-Diasca actor from another language may prove difficult, **there are settings where by design less leeway is granted** to at least some actors, in which case the integration of third-party code and the development of relevant APIs could be significantly eased.

A typical use case is the one of the **dataflow**, where the dataflow actors are quite constrained in terms of interactions (that have to be mediated through input and output ports). As a result, a CS2-level coupling API could be considered in that case, please refer to the A Focus on the Special Case of the Dataflow section for that.

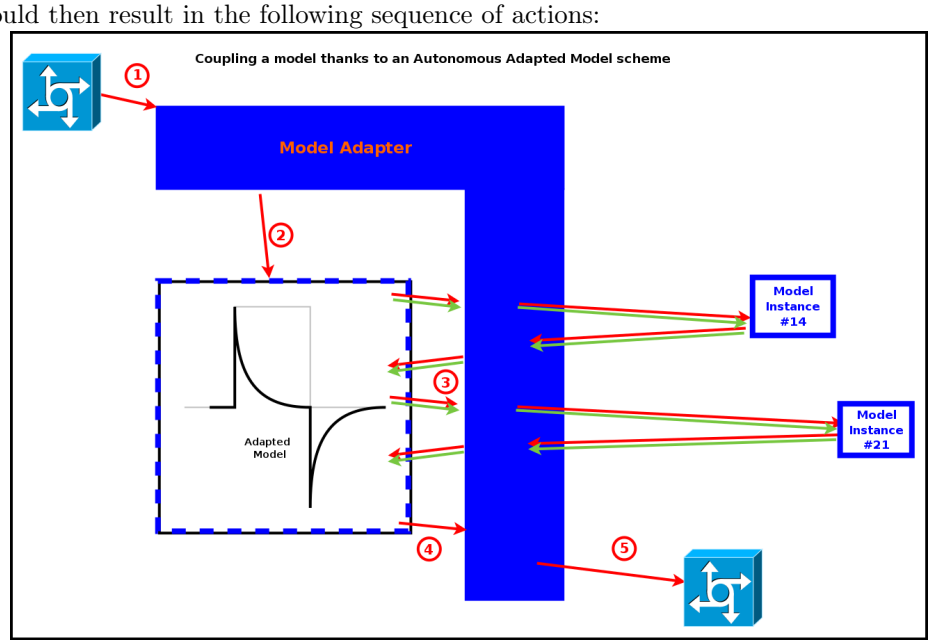
More generally, it shall be noted that in this CS2 scheme, for each of these overall logical adapted models, there are:

- an operating system process, dedicated to the adapted model itself; this would typically be a UNIX process executing a Java virtual machine, or a Python interpreter, etc.; these external system-level processes would appear as they were Erlang nodes, so that they can communicate with the

rest of the simulation (refer to the Interworking with Other Programming Languages section for more details)

- an Erlang (lightweight) process, animating the model adapter that itself encapsulates the adapted model, so that the whole behaves like a Sim-Diasca actor

In any case, as shown in the diagram below, an evaluation of the model would then result in the following sequence of actions:



1. the **model adapter**, once having to be scheduled, is triggered by the engine

2. **the adapter then hands over the actor's control flow to the adapted model**, which starts its processing

3. the adapted model will most probably require and/or produce pieces of information; the corresponding fetching and sending would be directly decided and operated *by the model itself*, through the **coupling API exposed by the adapter**; these calls (any number of model-triggered exchanges may occur) would result in interactions with other model instances, the whole having to be seamlessly orchestrated by the engine whereas it has no visibility anymore onto the course of the operations; moreover in the meantime other actors may have to perform parallel interactions with this adapted model - which it ought to be able to transparently handle (a problem with no clear solution)

4. once **the adapted model** has finished its operation, it then **relinquishes the actor's control**, which is transferred back to the adapter

5. the adapter notifies then the engine that the corresponding actor has finished its evaluation

Pros:

- from the point of view of a model maintainer, switching native inputs and outputs with their coupling counterparts is probably the lesser effort (at least one of the simplest to contemplate, if not to implement)
- this is very speculative, but a generic-enough coupling API could possibly hide the actual coupling solution being used (favouring their substituability, even if some common technical context would probably be implied)

Cons:

- not all programming languages can be accommodated (but, a priori, C, Java and Python could); yet one full set of APIs would have to be implemented for each supported language
- reversing the control flow - a model "driving" a part of the simulation - is less natural and is more likely to introduce issues (ex: non-termination of the model, cyclic dependency requiring the adapter and/or the adapted model to be reentrant, the whole being prone to deadlock); more generally, the engine would then have no real control onto the behaviour of the model, especially to the calls made through the coupling API; so, unless investing much efforts in the APIs, a misguided model might make a simulation yield wrong results and misbehave in unanticipated ways; moreover masking the consequences of the reversed control flow when developing the coupling API may be quite an endeavour - if achievable at all (possibly *not* in the general case)
- the adapted model may need information from other model instances but, conversely, other actors may require information from it; this feature being by design absent from the original model, its adapted version shall probably have to support it explicitly (as its state cannot be guaranteed to be up to date and consistent at any time); the adapter being unlikely to be able to provide this (as in this scheme the model state is kept private to the adapted model), the required modifications onto the model (and to its adapter as well) would be most probably significant and difficult to operate

2.3.3 Adapter-Driven Model [CS3]

This coupling scheme may superficially look similar to the previous one; it is actually quite different. Indeed in this scheme **the model appears to be mostly made of the adapter itself**, since most its key elements, from the state to the interactions, are actually directly mediated by the adapter itself; **however the actual computations are - at least to some elected extent - deferred to the actual, embedded model, which can be seen here mostly as a domain-specific library**, i.e. as a set of exposed functions.

So the role of the adapter is here to:

- maintain the model state
- drive the behaviour of the model:
  - regarding the engine, for scheduling and interactions
  - regarding the other models, to trigger actor messages on them and, reciprocally, handle the incoming ones, so that the information relevant for that model can be retrieved and propagated
  - by transparently delegating at least a part of the actual work (typically the domain-specific processing) to the embedded model, i.e. deciding which of the functions exposed by the model shall be called, when and with which parameters

By moving the centre of gravity (with regard to control, state, interactions) from the embedded model to the adapter, **the tricky part of the coupling, which is the synchronisation with the rest of the simulation, can be secured directly and a lot more easily; moreover the degree of integration can be finely tuned**.

Indeed embedded models could expose only a very limited set of functions (possibly just one in the simplest cases, a function that could be named for example `process` or `compute`) to perform their actual operations, while, over time, the services they offer might be subdivided into finer and finer pieces, for better control and selective interactions[6]; the granularity of the computations exposed by the embedded model can be freely adjusted to accommodate the interactions needed by a simulation, which can be seen as well as finding a balance between the operations delegated to the model and the ones directly taken in charge by the adapter - knowing that being able to change their respective amount over time could be very convenient.

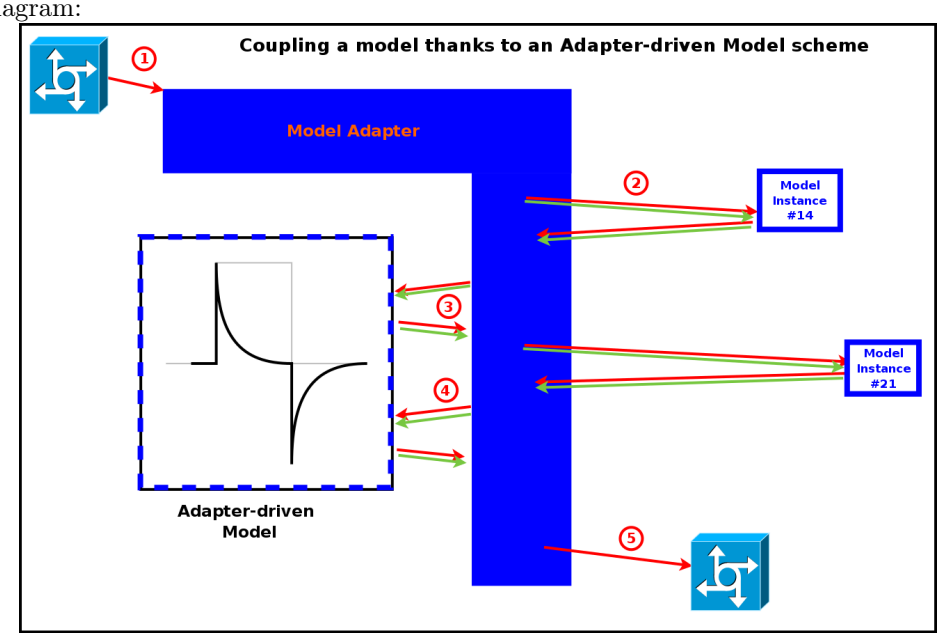
In some cases it may be contemplated that the state of that actor is split in two parts:

- the "public" one, exposed to the other actors (of course through relevant actor messages), and kept in the adapter
- the "private" one, invisible for all but that instance of adapted model, a state that it would keep internally

[6] For example in some cases the precise set of needed data cannot be determined from the very start of the model's evaluation at a given timestep; hence, instead of first collecting inputs, then fully performing the processing, then collecting the outputs, it could be more effective, thanks to the adapter, to intersperse more finely the computations with the relevant data exchanges.

Then the public state may be minimized in favour of the private one, native in terms of programming language for that model.

The series of operations for this coupling scheme is illustrated in the next diagram:



1. the engine triggers the **model adapter**, seen as a standard actor
2. the adapter knows by design which model-level embedded services shall be called, and thus what data shall be fed to them; as a consequence **the adapter uses actor messages to obtain these information from other actors**
3. having secured the first relevant input data, **the adapter is then able to call the intended services** offered by the embedded, adapter-driven model, and to get back their results, similarly propagated through actor messages
4. **any number of such exchanges and calls** can be performed to get data, execute services and update back the simulation, until the whole processing is done
5. finally the **adapter notifies the engine** that the evaluation of this actor is over

Pros:

- most of the "public" model state lies in the most practical place, the adapter, where it can easily be updated and accessed to from the rest of the simulation
- defining a clearly separated model-level library exposing well-identified services can favour capitalisation and re-use, and allow for an easier testing of it, in isolation

- reciprocally, a mock model library, whose API is implemented first only thanks to dummy terminators, can be used to uncouple the macroscopic interaction logic (i.e. the adapter) from the microscopic, domain-specific one (the embedded model, which may not be available from the start)

Mixed:

- while allowing a behaviour close to the one of a black box (in the sense that the implementation of each of the functions exposed by the model might remain opaque), similarly that code shall be trusted, studied or, ideally, sandboxed
- depending on its actual implementation (either as a shared library or as a separate system process, see below), the embedded model may rely, from a timestep to another, on a private, internal state of its own, which would remain unbeknownst to the engine

Cons:

- depending on the precise technical approach being used, the stability of the model may affect the one of the simulation (i.e. if the model crashes, it may crash the simulation with little information reported)
- a partly language-specific interconnection scheme must exist; typically the embedded model may be implemented as a shared library (ex: `.so`, for the most common C-like linkage, thanks to NIF or linked-in drivers, with little to no extra process involved, and no private state) or, for some languages, higher-level, more integrated solutions (involving generally a separate system process), generally inducing more overhead, exist (refer to the Interworking with Other Programming Languages section for more details)

> **Note**
>
> This coupling scheme and the previous one may be seen as special cases of a more general scheme, where an applicative protocol, relying on any kind of IPC, may couple the logical processes that correspond respectively to the adapter and to the model in relations that are more complex than having a master and a slave.
> However considering the adapter and the model as peers does not seem to grant specific benefits, and surely leads to a more complex design, hence we retained here only strict master/slave relations, with CS2 and CS3.

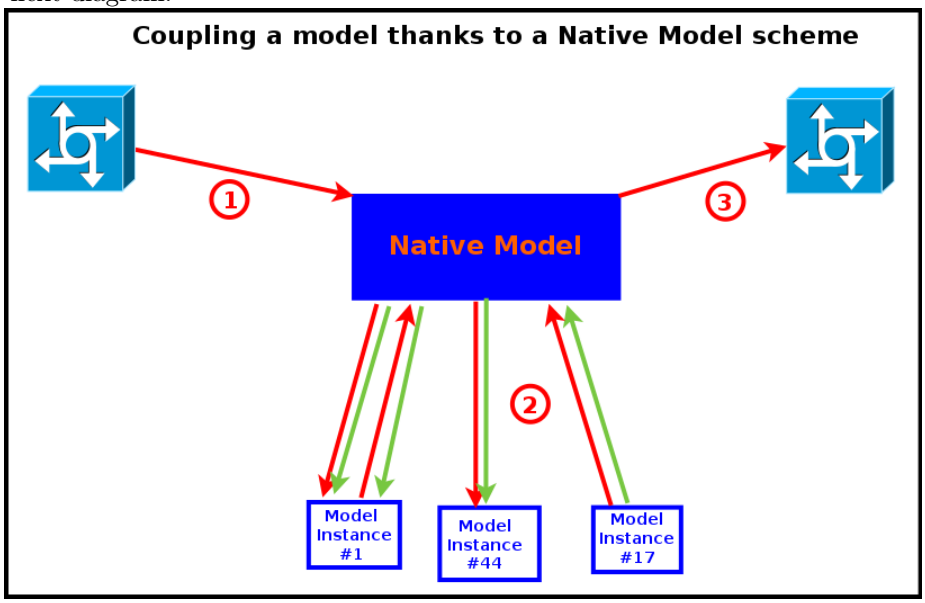
2.3.4 Native Model [CS4]

This coupling scheme is the most straightforward one: like for most simulations, a model here is specifically designed according to the engine that is supposed to evaluate it.

As such, the model is fully compliant with the framework, and is perfectly integrated with the rest of the simulation, which often leads to the best controllability and runtime performances (significantly better than the ones obtained with co-simulation).

Of course developing a model specific to Sim-Diasca has a cost; however integrating a third-party model requires efforts as well, and they may be significant.

The series of operations involved in this coupling scheme is simple, as it corresponds exactly to the normal mode of operation of the engine, as shown in the next diagram:



1. the engine triggers the **native model**, which *is* an actor like any other standard one
2. this **actor is then free to interact at will with the other actors** (exchanging information, performing computations, etc.)
3. then the engine is notified that **the evaluation of this actor is over**

Pros:

- best integrated solution, inducing a simple, well-supported, quite transparent and efficient setting
- only solution (with some versions of the adapter-driven one) not involving an overhead of one operating system process per instance of coupled model - hence expected to be significantly more scalable
- quite close to the adapter-driven coupling, hence allowing to precisely tune how much of a model is to be put directly in the simulation (as in this CS4

scheme) and, reciprocally, how much of it should be placed in a separate library (as in the CS3 scheme)

Cons:

- implies that the model is developed in Erlang, a language which is most probably less familiar to model maintainers than others; however the role of domain experts might not overlap the one of developers anyway (all programming languages, including Java and Python, require a sufficient amount of knowledge in computer science)

2.4 A Focus on the Special Case of the Dataflow

Some simulations significantly deviate from the usual multi-agent, disaggregated scheme often seen in the simulation of complex systems, and are best expressed according to alternate paradigms.

One of these paradigms is the **dataflow**[7] architecture, where operations are implemented by blocks that are interlinked in a rather static overall assembly.

This architecture, involving fixed, typed routes delimited by input and output ports, is more constrained than the usual multi-agent approach, and as such **offers coupling perspectives of its own**.

Indeed, even if of course the targeted dataflow may involve only Sim-Diasca actors, having to dispatch the corresponding computations to well-defined blocks fuels the possibility of having some of these blocks be implemented as third-party components, with their own technical conventions and languages.

We therefore identified specifically two main *coupling* opportunities pertaining to the dataflow approach, discussed and contrasted below: having library-based blocks, or blocks using a foreign dataflow API.

2.4.1 Library-Based Blocks

This architecture simply corresponds to CS3 (i.e. the *Adapter-Driven Model* coupling scheme) once applied to the dataflow special case.

More precisely, in this approach a given third-party model is structured as a domain-specific component offering a set of pre-implemented computations, of relevant granularity. **On top of this "expert library", a Sim-Diasca dataflow adapter is then defined**, in order to obtain from it an actual dataflow block.

The purpose of this adapter is to drive that embedded domain library in the context of the underlying dataflow, triggering the services it offers according to a relevant logic, feeding it with the proper information and fetching from it the relevant outputs.

Taking the example of an hypothetical model of a heat pump, the third-party library may split related computations into, say, 5 functions allowing respectively to determine:

- whether all the conditions for the heat pump to be successfully switched on are met
- what its operational efficiency under a given context is
- what are the various operating steps the pump may go through
- what are the corresponding use of the heat source and external power, and the consequences onto the heat sink
- what are the likeliness of various faults to happen

These domain services exposed by the library, which are often at least loosely coupled, might then be federated by a relevant Sim-Diasca adapter, in charge of:

[7] Please refer to the *Sim-Diasca Dataflow HOWTO* for more information.

- defining the relevant operations involving these services to account for the targeted computation block[8]
- requesting from the simulation (thanks to actor messages) the necessary information held by other actors that shall be supplied as input to these domain-specific functions
- getting back from them the corresponding outputs, using them to update the state of this adapter and/or the one of other actors, possibly also generating simulation results from them thanks to probes

Of course a bridge must be created between the third-party library and the adapter. As in the general case different, non-Erlang programming languages have to be accommodated, please refer to the Interworking with Other Programming Languages section for the corresponding technical details.

An advantage of basing a block onto a third-party library augmented of an adapter is the clean separation between the two: the domain-specific part is embodied by the library while the technical, engine-specific, dataflow-compliant part lies in the adapter.

Doing so allows to define an autonomous library that can be used in multiple contexts, the one with the Sim-Diasca dataflow adapter being only one of them.

For example, on top of the same library, separate, re-usable tests can be defined, adapters to other platforms can be devised, etc., while the library can still be used directly, in ad hoc developments.

2.4.2 Blocks Using Foreign Dataflow API

Another coupling approach can be considered in such a dataflow context: any component can be safely integrated provided that it respects the laws and rules applying to the dataflow at hand.

For that a dataflow API must have been defined, and be implemented in the programming language that has been chosen in order to develop the model.

The model would then take the form of a dataflow block, and use directly the various constructs provided by the API, such as input and output ports, that are implemented by the coupling architecture - and all that developed in one's language of choice.

Using such a foreign dataflow API allows to stick to one's favorite language (ex: Python or Java), sparing the need of using Erlang - of course on condition that this API has already been implemented in that particular language.

A drawback is that this approach results in a version of the model that is dedicated to this coupling architecture, and thus cannot be readily used in other contexts.

2.4.3 Choosing a Dataflow Option

The choice between dataflow coupling options is to be made *per-block*: in a given simulation, native dataflow blocks (i.e. based on classical Sim-Diasca actors), library-based ones (based on Sim-Diasca dataflow adapters) and blocks relying

[8]The logic of a block may exceed a plain, unconditional series of steps; in the general case an adapter-level algorithm is to be defined (ex: with stateful logic, conditional sections, loops, etc.).

on one of the foreign dataflow APIs can safely coexist, as they obey the same conventions.

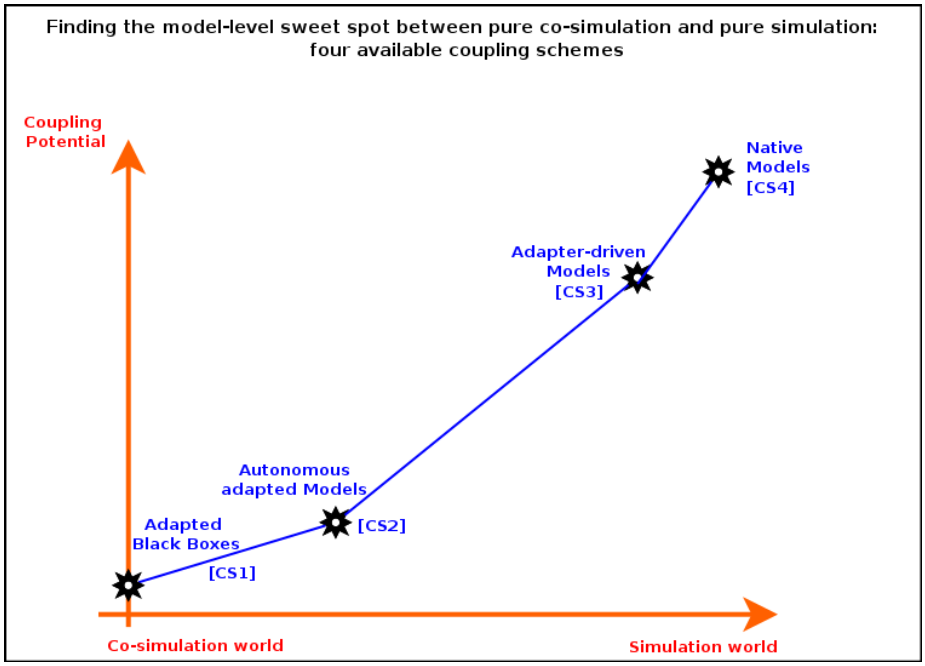
So, for a given model, how should it best integrated as a computation block in a target dataflow?

If a clean separation between the domain expertise and the software development is sought upon, then the library approach shall be considered, especially if technical support is available in order to couple the model (i.e. write the dataflow adapter); this would offer much potential for model re-use.

Conversely, if developing in Erlang is a problem, and if, for a target language, the dataflow API is already available, then this route should be favored, especially if the model is to be written for this sole context.

2.5 Recommendations in terms of Choice of Coupling Scheme

2.5.1 Sweet Spot in terms of Coupling Schemes



2.5.2 A Very Basic Decision Tree to Choose a Coupling Scheme for a Given Model

> **Note**
> As mentioned, a different coupling scheme can be chosen *for each model*.

Is this model is to be included in a dataflow approach? If yes, refer to Choosing a Dataflow Option.

If no, is this model delivered as a black box, or is it overly complex?

- if yes, use **CS1**, and develop an input and an output translators
- if no, is this model simple enough so that a full conversion to Sim-Diasca can be considered?
  - if yes, use **CS4**, and make a native Sim-Diasca version of it
  - if no, is the model's main programming language among the ones (see list) that can be easily interfaced with?
    - if yes, use **CS3** and split the processing performed by this model into chunks that can be appropriately called by the adapter
    - otherwise: use the default scheme, **CS4**, and develop a native Sim-Diasca model

(as a result, due to its limitations, we currently see no use case where CS2 should really be specifically recommended)

2.6 Other Coupling-Related Topics to Consider

Ideally, regardless of the coupling scheme being chosen, following topics should be secured in the context of the finally obtained implemented model.

2.6.1 Simulation Results

Results shall be sent from models; of course the actual mode of operation would depend onto the coupling scheme:

Coupling Scheme	**Mode of operation for results**
CS1	Standard probes can be directly used by the output translator.
CS2	A model-level additional API could be offered, relying on underlying probes (standard or specialised).
CS3	The adapter would directly process and send the results through standard probes.
CS4	Standard probes can be used, or any specialised ones that would be needed.

2.6.2 Error Management

Regardless of the coupling scheme, errors should be detected as early as possible, reported with an informative message describing at least a relevant context, and should in the general case halt the simulation ("*fail fast and loudly*"; of course silent failures shall not be tolerated).

Sim-Diasca has been designed so that no model-level error could remain unnoticed; on the contrary, measures have been taken so that the crash, the faulty behaviour or the seemingly non-termination of a model instance will halt the simulation on failure (regardless of parallelism and distribution) with a detailed error report, the engine attempting to establish a more precise diagnosis.

How errors shall be reported weakly depends on the coupling scheme:

Coupling Scheme	**Mode of operation for error reporting**
CS1	The adapter is able to report errors should either of the translators or the execution of the black box fail (at least error codes shall be obtained in this latter case, possibly core dumps could be generated and stored for post-mortem analysis).
CS2	The adapter can directly report errors, while a dedicated error management API should be offered to the adapted model.
CS3	The adapter can directly report errors, based on the error status reported by the calls delegated to the embedded model (a convention transverse to all models ought to be taken there).
CS4	The actor would also directly use the error API, like all actors.

2.6.3 Simulation Traces

These runtime traces, which can be purely technical and/or relative to the domain-specific mode of operation of the model (possibly on separate channels), are priceless to troubleshoot many design and implementation issues.

They shall be reported by the model based on the support provided by each coupling scheme; these traces would be aggregated by the native Sim-Diasca concurrent trace system, among any other system(s).

Coupling Scheme	Mode of operation for trace sending
CS1	Traces from the black-box model would typically have been written in a model-specific file according to a format that would be parsed by the output translator and forwarded, possibly enriched, through the Sim-Diasca trace system, among the other traces emanating from the adapter.
CS2	An additional model-level API could be offered, linking to the engine trace system.
CS3	Either traces would be returned by the library functions alongside the result or, possibly, an adapter-level function pointer counterpart, corresponding to a trace sending primitive (targeting both systems), could be specified amidst their parameters.
CS4	Vanilla Sim-Diasca traces could be readily used.

3 Appendices

3.1 Interworking with Other Programming Languages

Often a separate process animated by the target programming language is spawned. Then following components are used:

- the Sim-Diasca adapter (in Erlang)
- the adapted model (in its native language)
- an adaptation bridge (still in the native language of the original model), mediating the technical bidirectional communication between the previous two components

3.1.1 Interworking with Java

So that Java code can interact with Erlang code, one should use Jinterface.

Parallel execution and bidirectional control and exchange can be implemented.

The adaptation bridge here is the thin Java layer added on top of the adapted model so that it can emulate the behaviour of an Erlang node (hence relying on the primitives offered by Jinterface in order to define a mailbox, send and receive messages, create Erlang terms, etc.).

The purpose of this bridge is mostly to route the requests emanating from the adapter so that their Java implementation is executed and their results are propagated back.

3.1.2 Interworking with Python

This can be done either thanks to ErlPort (probably the best choice), or to Py-Interface.

Parallel execution and bidirectional control and exchange can be implemented.

3.1.3 Interworking with Other Languages

Links known to exist:

- to `C`: use C nodes, based on an Erlang port
- to any language able to generate a dynamic library with C-like linkage (ex: `*.so`), like `C++`: use a linked-in driver (exchanging with it like for a port) or a NIF (calling directly functions from a library), with little to no overhead (typically extra process) involved
- to the `.Net` platform: use OTP.NET, through a port of the JInterface code
- to `Perl`: use Perl Erlang-Port, through a port
- to `PHP`: use PHP/Erlang to turn a PHP thread into an Erlang C node
- to `Haskell`: use the Haskell/Erlang FFI, enabling full bidirectional communication

- to `Ruby`: use ErlPort, otherwise erlelectricity
- to `Scheme`: use Erlang/Gambit
- to `Emacs Lisp`, use Distel
- to any shell, or even directly from the command-line: use `erl_call`

This topic is also addressed in the Erlang FAQ and in the Interoperability Tutorial User's Guide.

Some interworking solutions are quite low-level, they induce little overhead (ex: for linked-in drivers or for NIFs, no extra operating system process is needed) yet they may affect adversely the Erlang VM (should they crash, leak memory, block their associated scheduler or do not terminate), while others offer higher-level, more integrated solutions (for example to transform adequately and transparently the datastructures being exchanged).

In these latter cases, care must be taken about scalability. For example, depending on the number of integrated instances that are involved, too many system processes might be created or too many file descriptors may be opened, as for each port one process is spawned while one file descriptor is used for reading and one is used for writing.

Custom protocols over IPC (ex: files, pipes, UNIX sockets, Internet ones, etc.), based on the exchange of binary messages, can also always be devised. This would show a priori little advantage over any of the already available standard protocols, yet would allow to add support for virtually any language or exchange channel. Extending `ErlPort` could be another option.