Presentation of

Sim-Diasca

Simulation of Discrete Systems of All Scales

Friday, April 7, 2023

Contact: Olivier Boudeville
(olivier.boudeville@edf.fr)

More information:

- http://www.sim-diasca.com
- https://github.com/Olivier-Boudeville-EDF/Sim-Diasca

Major French Utility

EDF

Presentation outline: introducing Sim-Diasca

Sim-Diasca is a **concurrent** (parallel and distributed) **generic discrete-time simulation engine** aiming at **maximum scalability** (millions of complex model instances in interaction).

- Generic, domain-agnostic: can be applied to a wide range of large-scale discrete simulation targets, from ecosystems to vast IT infrastructures
- Typically suitable for simulations in the field of Complex Systems

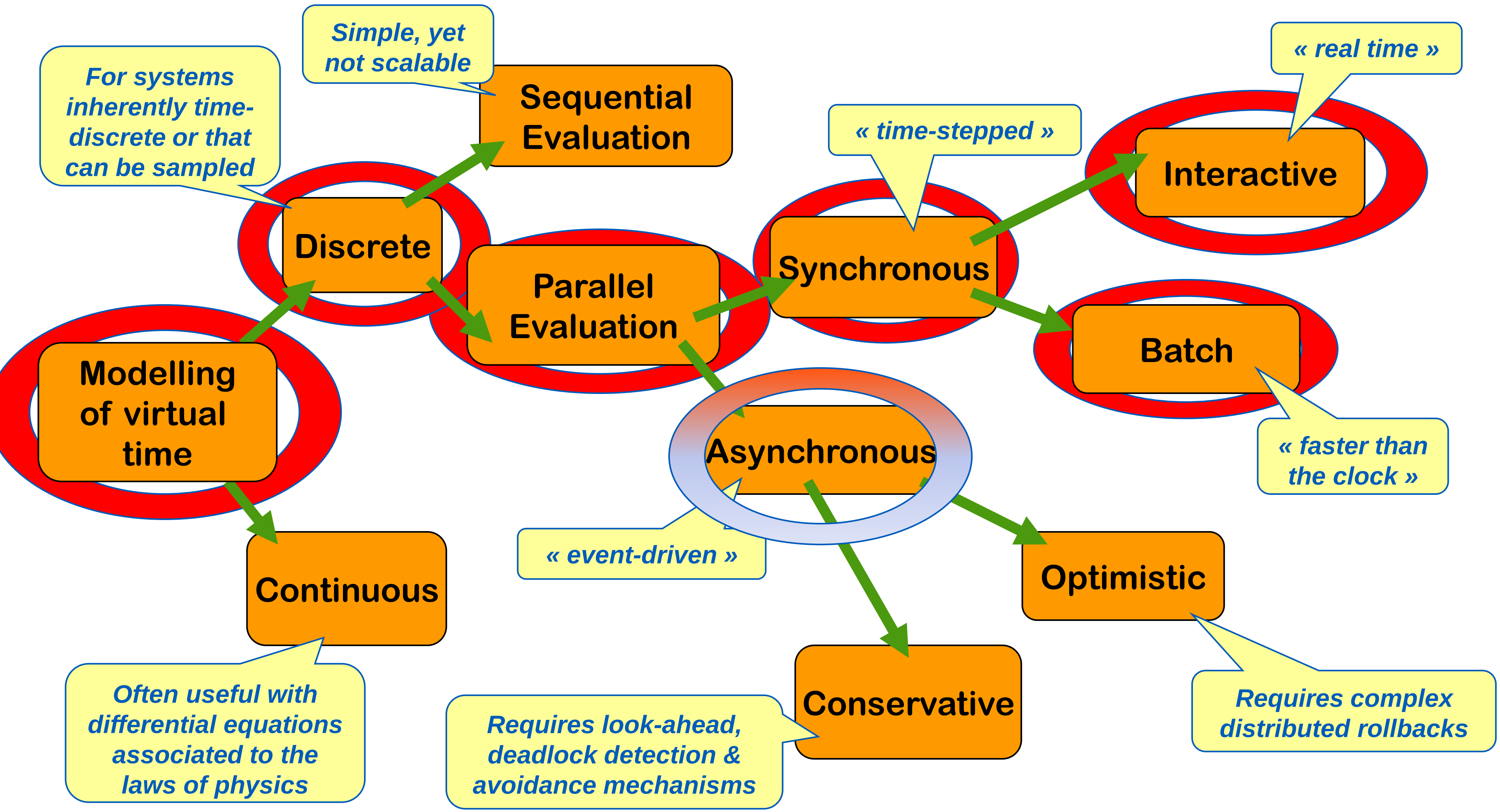
  (whereas most of the tools for that are sequential and can hardly scale)
- Sim-Diasca (simulation engine) + models + simulation case(s) ⇨ a simulator
- Fully implemented in a functional language, Erlang (http://erlang.org)
- Supported platforms: GNU/Linux (from single laptops to full-blown HPC clusters)
- Used by EDF and third parties, maintained by EDF R&D
- Released since 2010 by EDF R&D as free software (LGPL licence)

Sim-Diasca requirements & technical answer

Functional Requirements:	Technical Answer:
Multiple **usages** anticipated (simulator/emulator/digital-twin)	**Batch** or **Interactive** mode of operation
Need for **correctness** in the evaluation of all kinds of models	Respect of **causality** Support for **stochastic** models
Simulation use cases require to be able to: - **Replay** at will any given trajectory of the target system - **Correlate** directly a change into the simulation results to a change into the inputs - **Explore** all possible trajectories of the system, moreover in a fair, representative way	**Total reproducibility** Support of a certain form of "**ergodicity**", i.e. a guarantee that: - All possible outcomes according to the models *can* actually occur in the simulations - And that their probability of showing up in the simulations is close to the one that can be deduced from the models
Need to be able to simulate **very large systems**, potentially involving dozens of millions of interacting actors	Ability to **scale up** significantly: - At the algorithmic level: maximal parallelization of the evaluation of models - At the level of computing resources: harness multicores, SMP, clusters and other High Performance Computing solutions

Sim-Diasca main algorithmic choices

The nodal point is how the simulation time is managed:



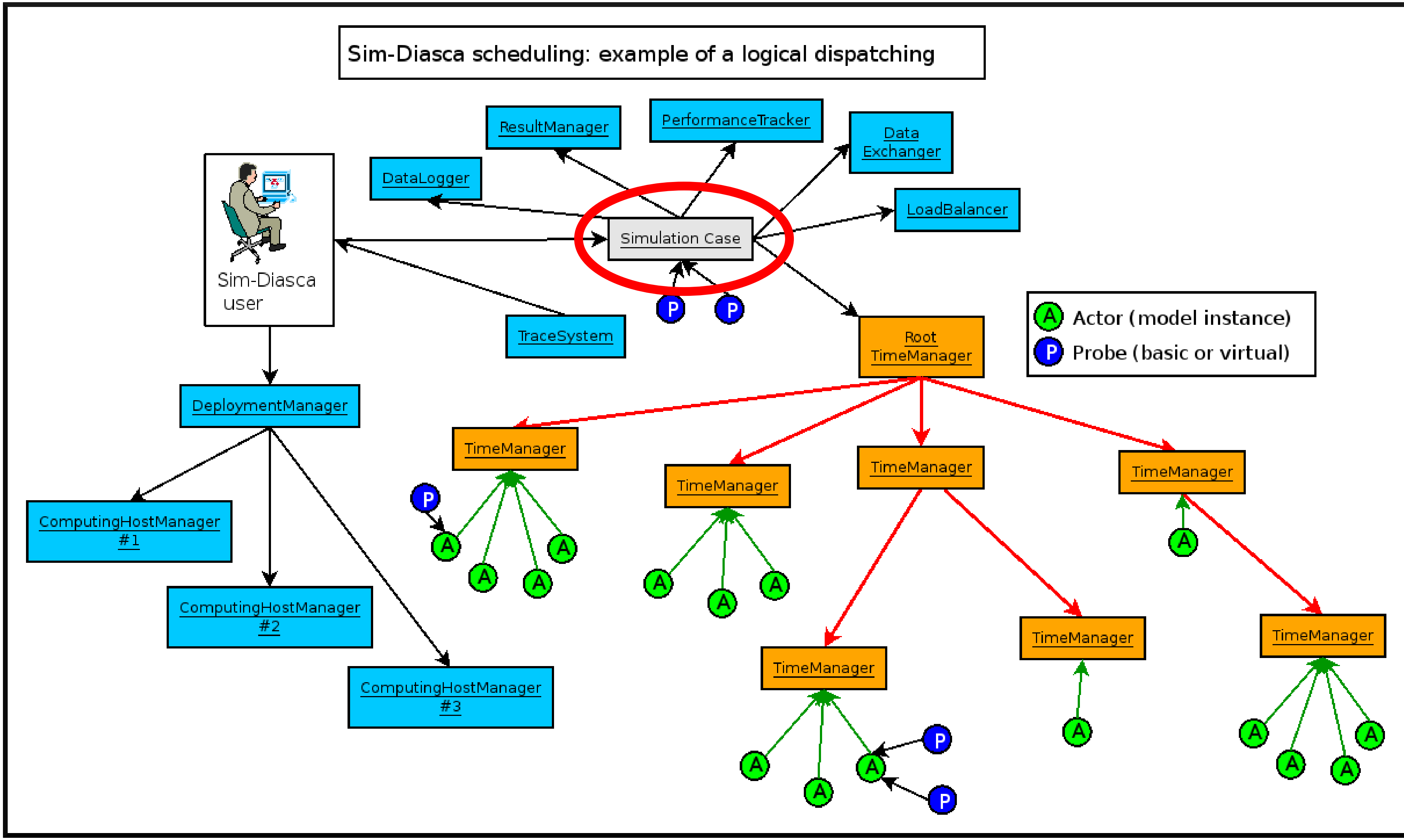
- Choices for the Sim-Diasca mode of operation are shown within red ellipses.

Sim-Diasca technical design & features

- As an answer to the requirements, a simulation engine:
  - **Based on discrete events:** the elements of the target system (model instances, a.k.a. actors) exchange messages and update their state accordingly
  - **Synchronous** (« *time-slicing* », « *time-stepped* »):
    - A fundamental simulation frequency is defined (by default, 50Hz)
    - In interactive mode, the engine adjusts its time steps (ticks) to the real (wall-clock) time
    - In batch mode, the engine processes its ticks at maximum speed, and jumps automatically over periods without any possible activity of actors (quite similarly to asynchronous approaches)
  - **Intensely concurrent**:
    - Distributed simulation: a single simulation can spread over a set of computing hosts (e.g. HPC cluster)
    - and Parallel, i.e. taking advantage, for each computing host, of all cores of all processors
    - The algorithm allows, at each scheduled logical moment, to evaluate all model instances *in parallel*! (And only the relevant ones)
    - Scalability-wise, at the end of 2010, the threshold of 1 million instances of rather complex models could be reached
  - **Granting a large freedom and expressivity to models**:
    - Modelling: object-oriented approach, whence implementation directly derives, based on a concurrent high-level functional language (Erlang)
    - Flexible and powerful scheduling policies for actors (fully passive, periodical, or driving their own behaviour arbitrarily)
    - Stochastic support: any number of stochastic variables per actor, respecting built-in or model-defined probability density functions
    - Very few constraints apply to models (e.g. no pre-established fan-in/fan-out, no look-ahead needed); no causality-induced time biases (as many logical moments - « diascas » - as needed will be created in a tick to sort out causality)
  - **Providing all needed features to build easily a simulator:**
    - Engine features: automatic and parallel deployment (code and data), management of results, load-balancing, integration to most platforms (e.g. clusters), distributed trace system, tuning for scalability, etc.
    - Model features: automatic message reordering, stochastic support, probe support (autonomous and database-based)

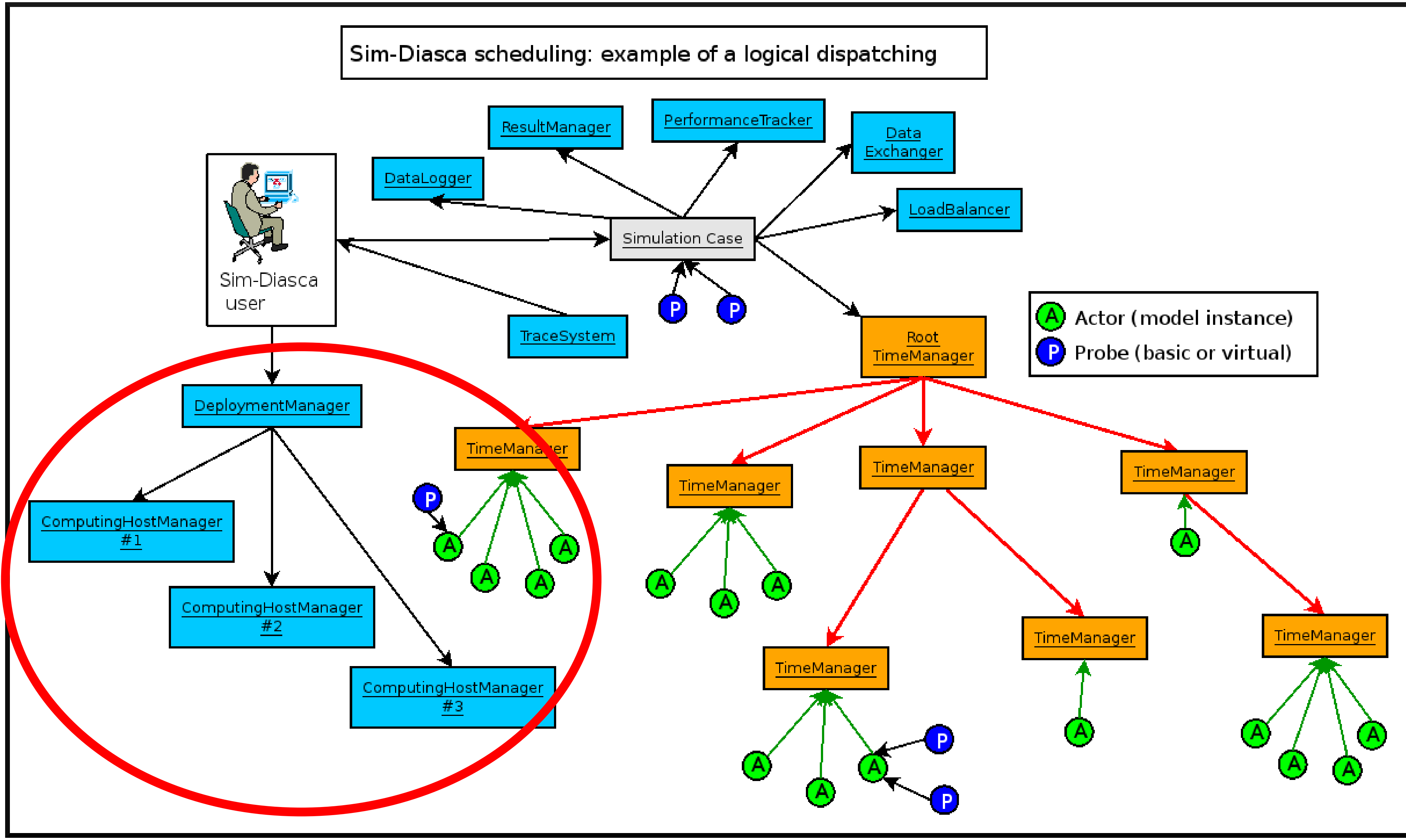
- Central point: the massive parallelism is achieved without prejudice to the targeted simulation properties

Functional Services: 1: Read configuration information

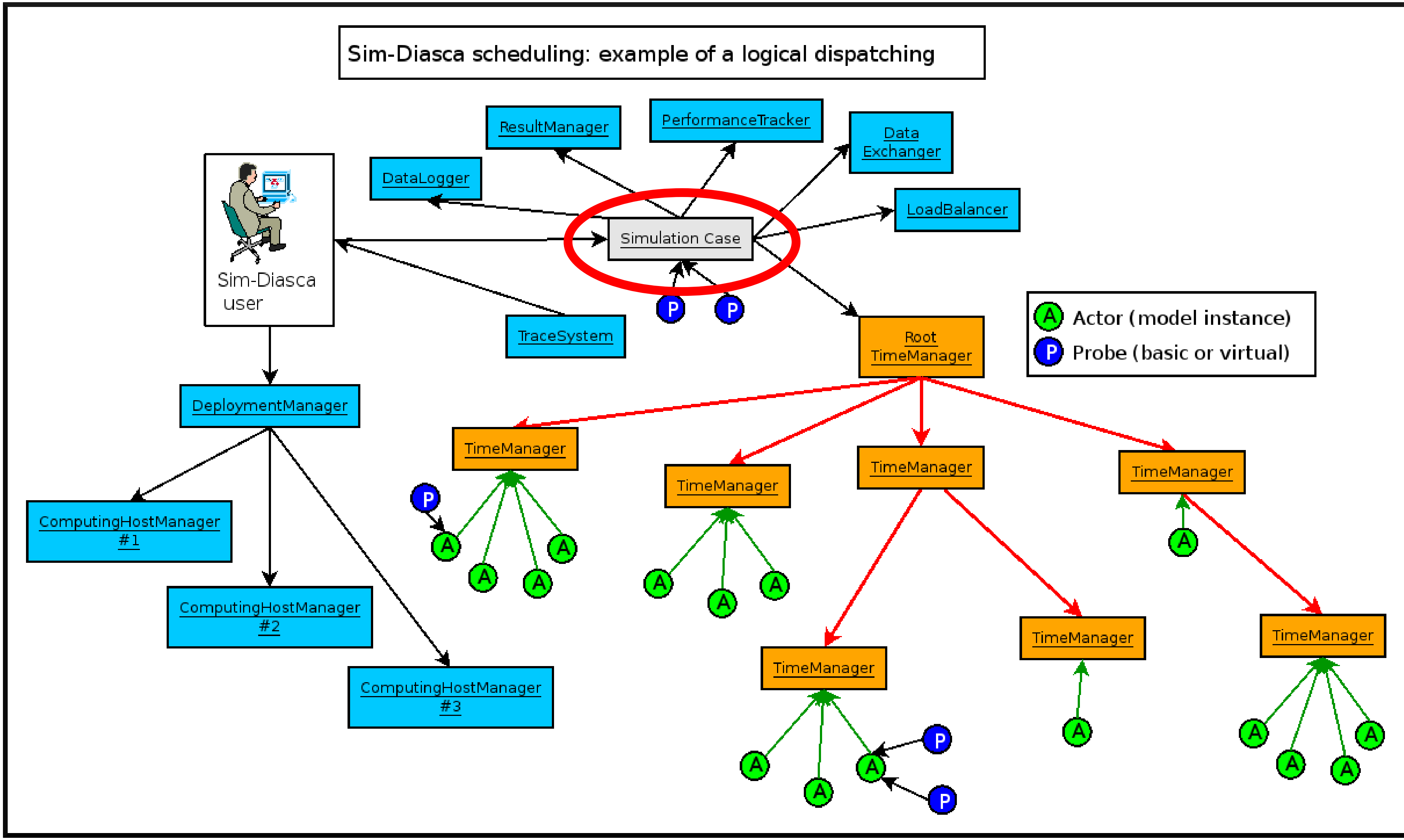


Functional Services:

2: Deploy simulation on hosts (no prior install needed)

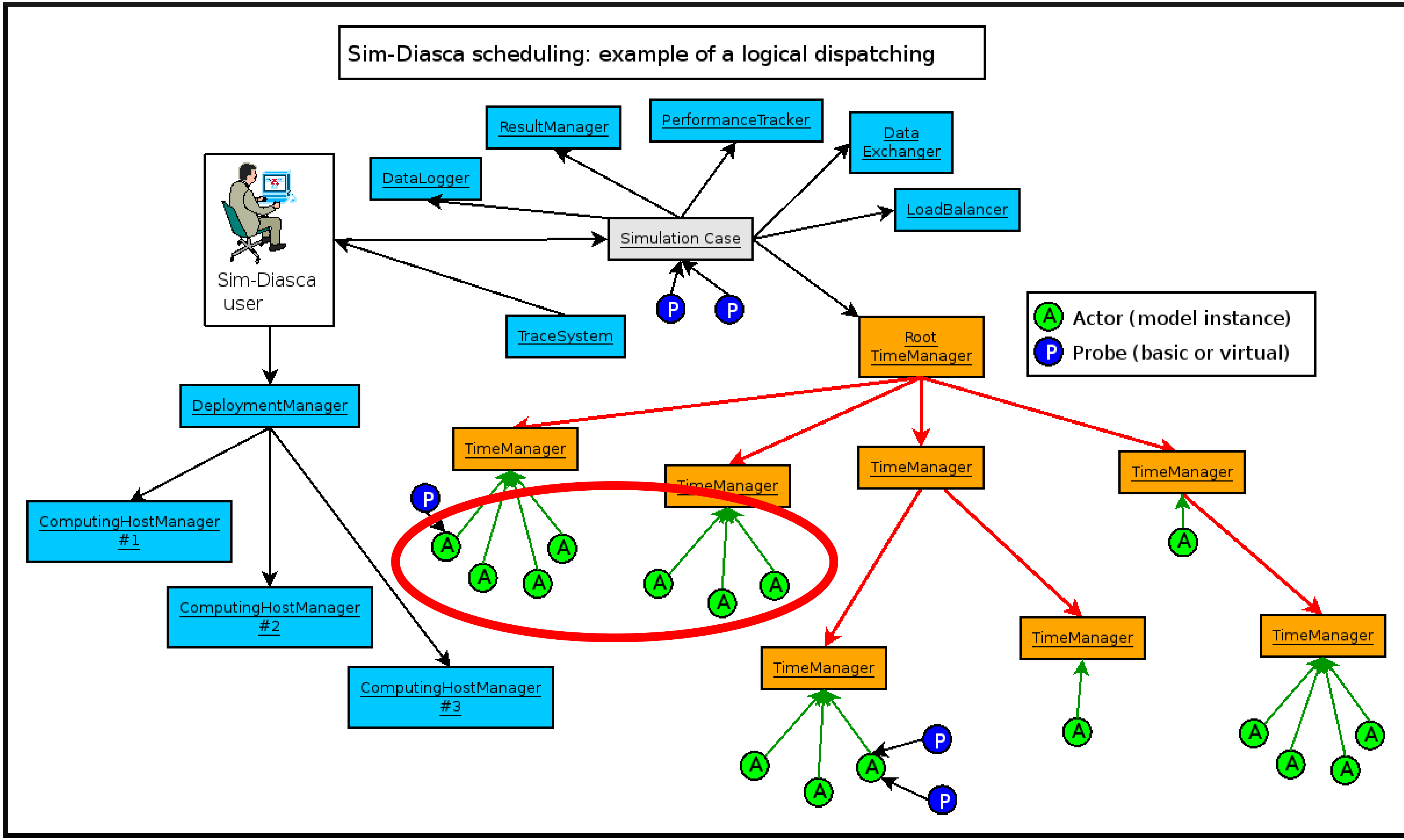


Functional Services: 3: Process simulation case

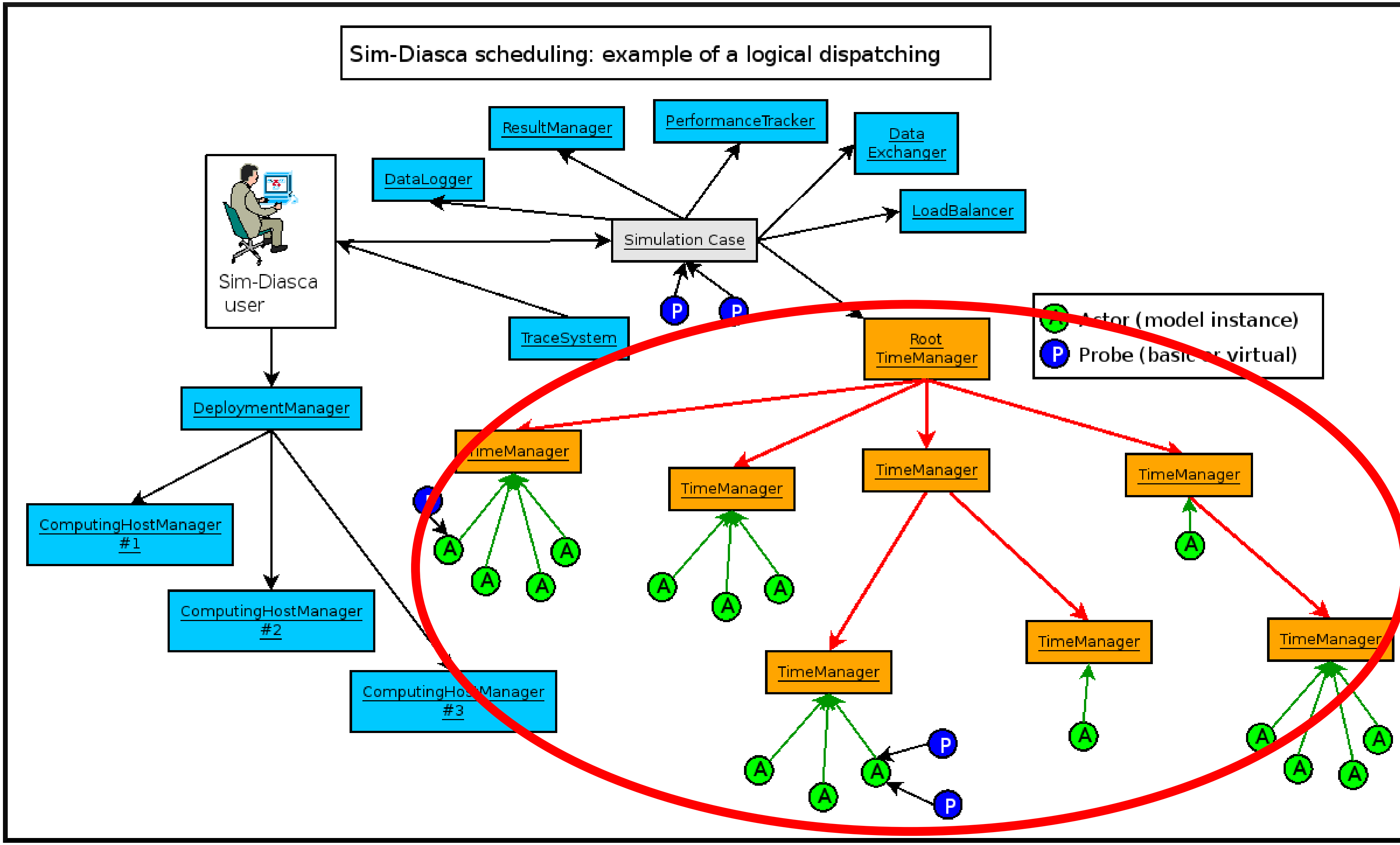


Functional Services:

4: Place and create initial model instances



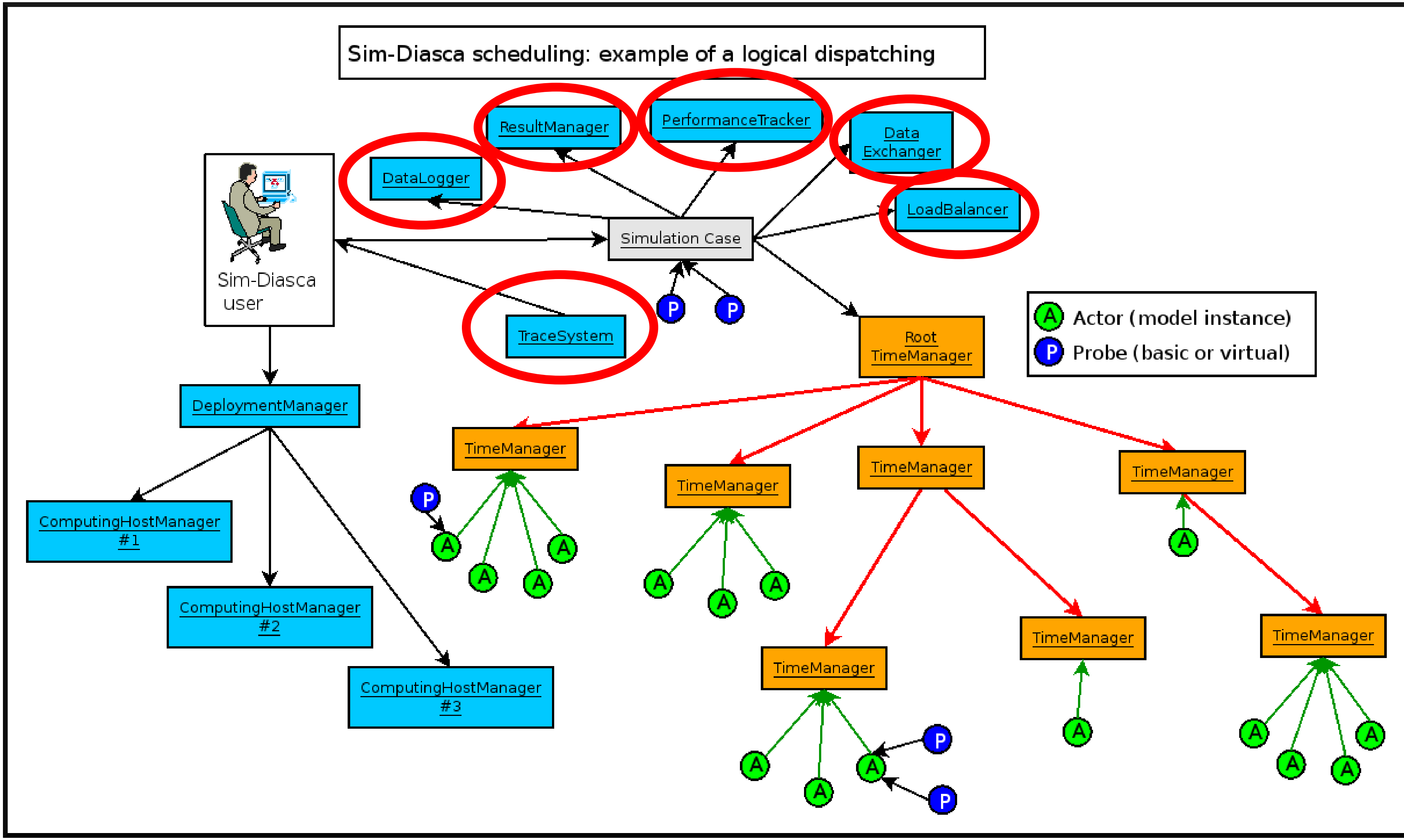
Functional Services: 5: Schedule model instances



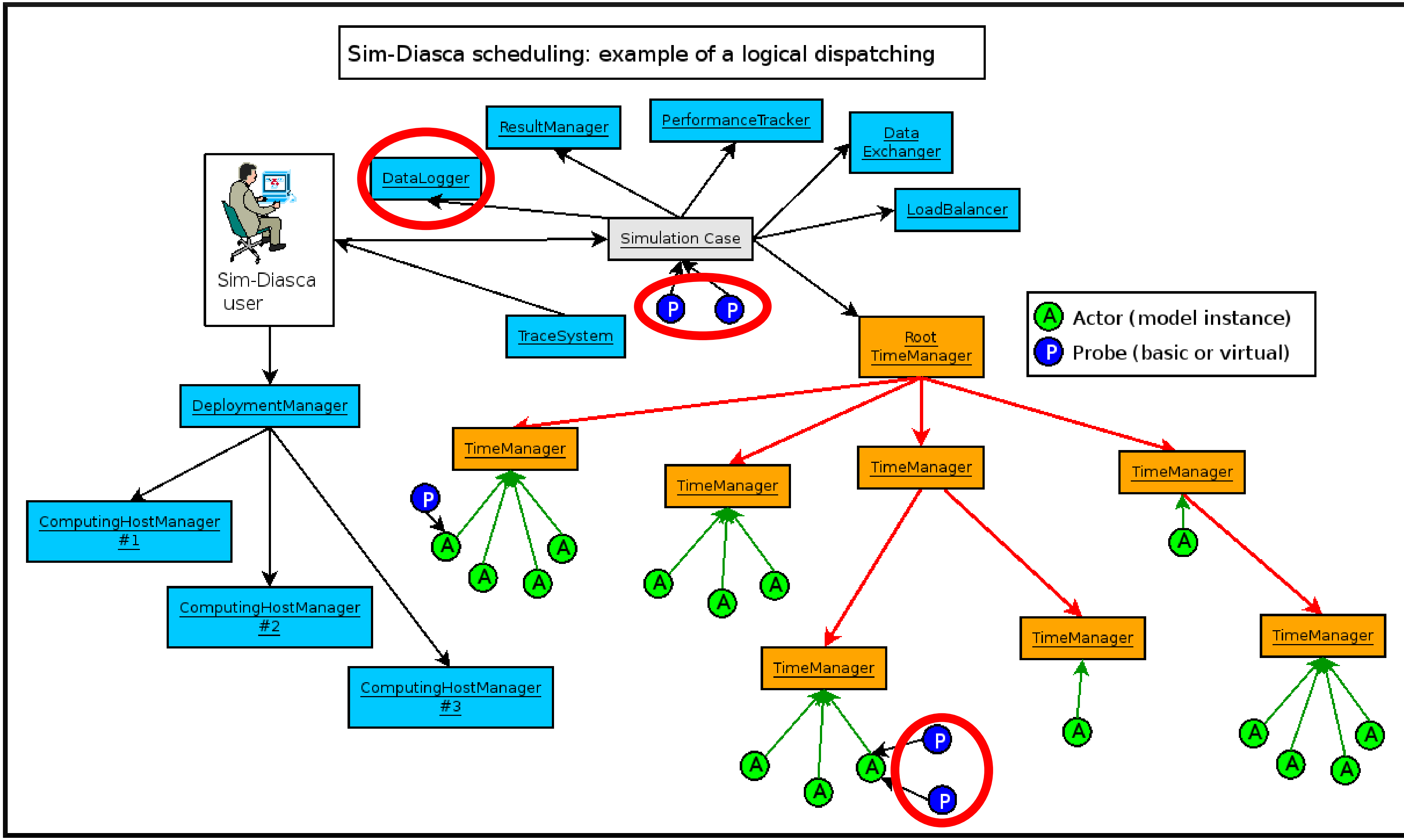
Functional Services:

6: Offer additional services

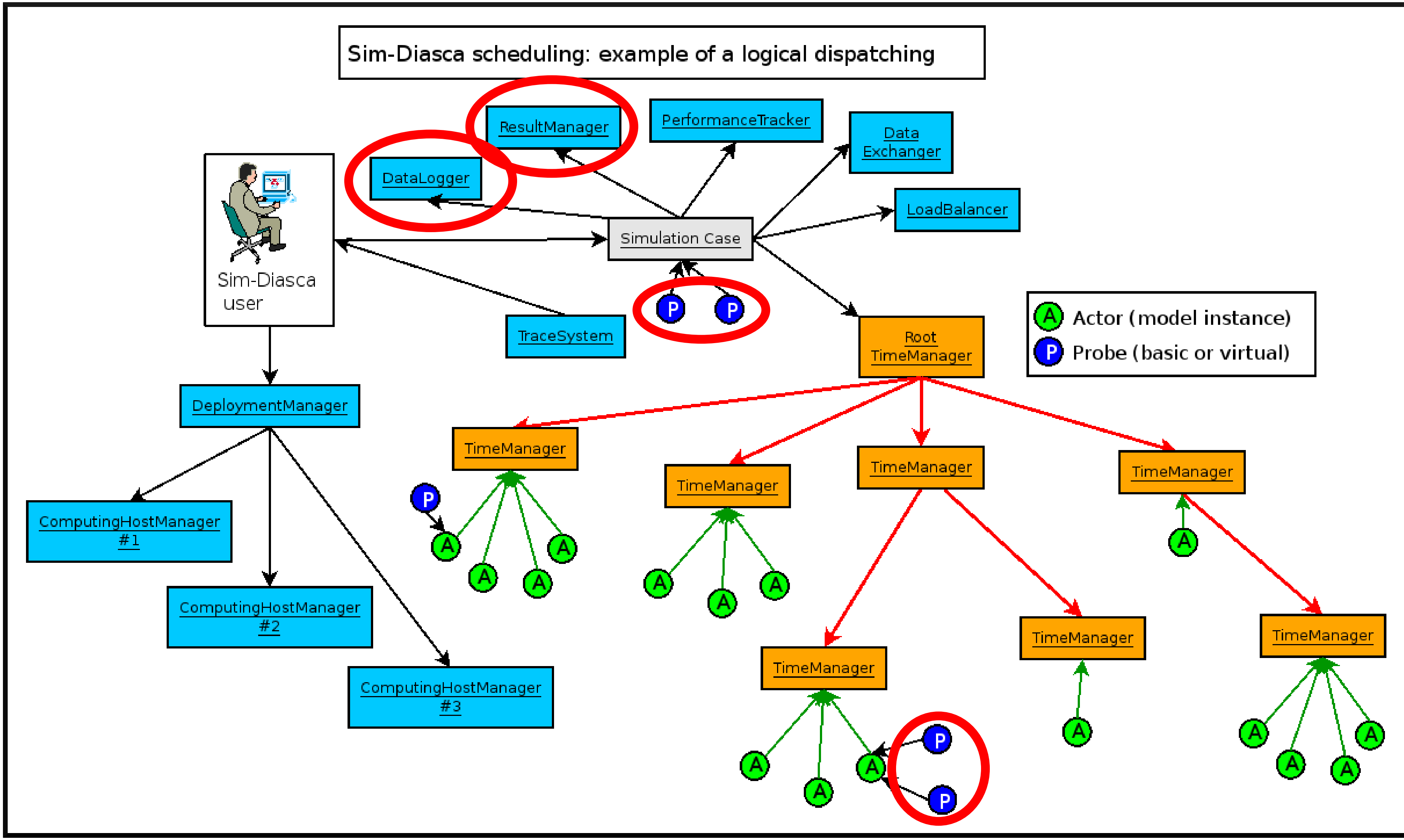
(e.g. distributed trace system, stochastic support, performance tracker, data-exchanger)



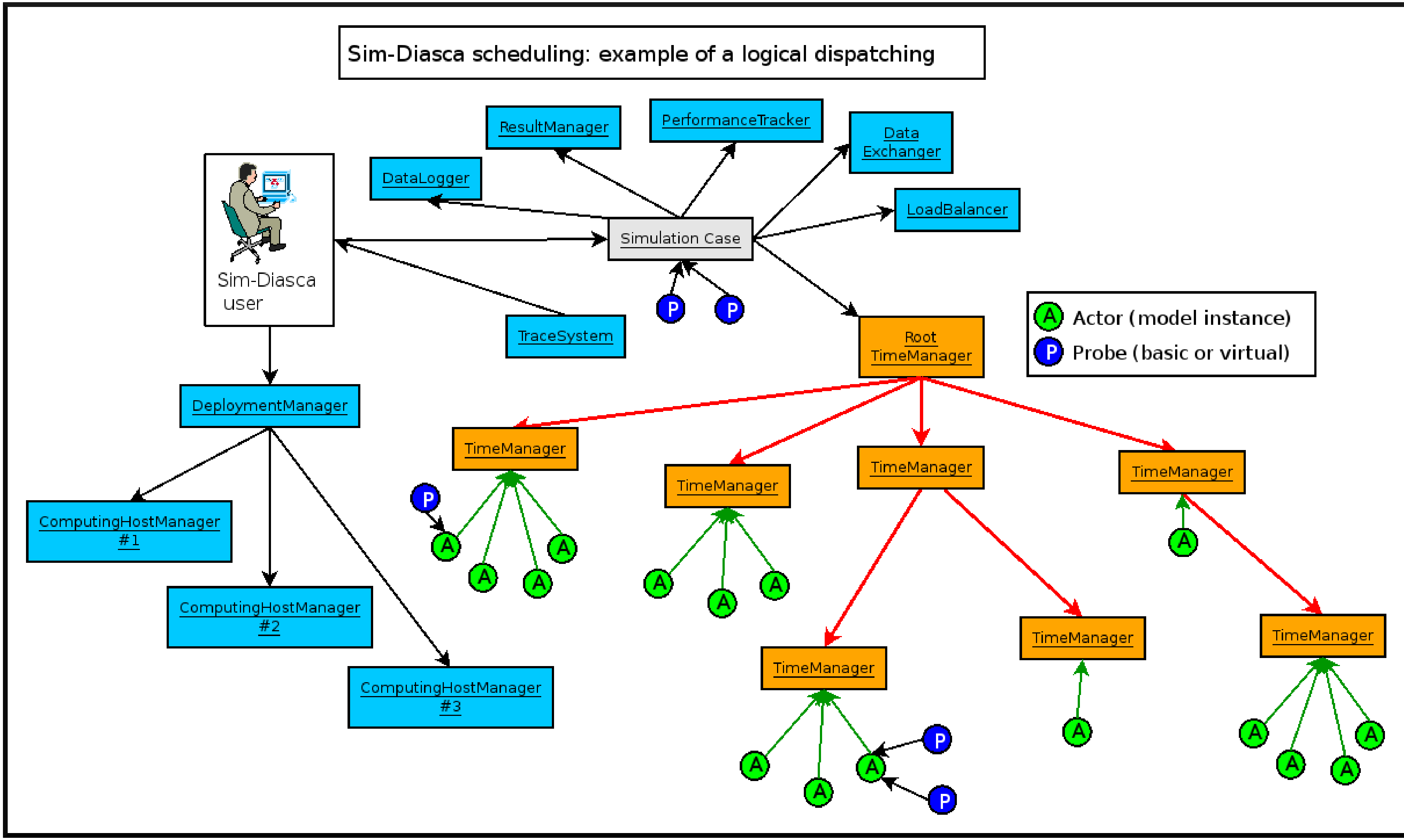
Functional Services: 7: Generate simulation data (e.g. time series)



Functional Services: 8: Select, generate and retrieve results (e.g. plots)



Functional Services: (post-processing skipped)



Sim-Diasca Key Points

- The engine must enforce notably following simulation properties:
  - Respect of causality
  - Total reproducibility
  - Some form of ergodicity
- The challenge is to obtain these properties in spite of massive parallelism and distribution
- To do so, inter-actor messages have to be appropriately reordered (transparently done by the engine)
- We target maximum scalability: potentially millions of instances of rather complex models can be evaluated in parallel
- Another technical challenge is to simplify as much as possible the model development:
  - **Bridge the gap** between model formalisations and actual simulation code (UML sequence diagrams, flow maps, state machines, dataflows): mixing programming styles (Object-Oriented and Functional ones)
  - **Provide stochastic support**
  - **Shelter models from parallelism**: write each of them in a simple, purely sequential way (and support a few bindings)
  - **Provide an higher-level language** with appropriate constructs so that domain experts have a better chance of understanding and developing actual models

More precise functional and technical requirements are detailed in annex 2.

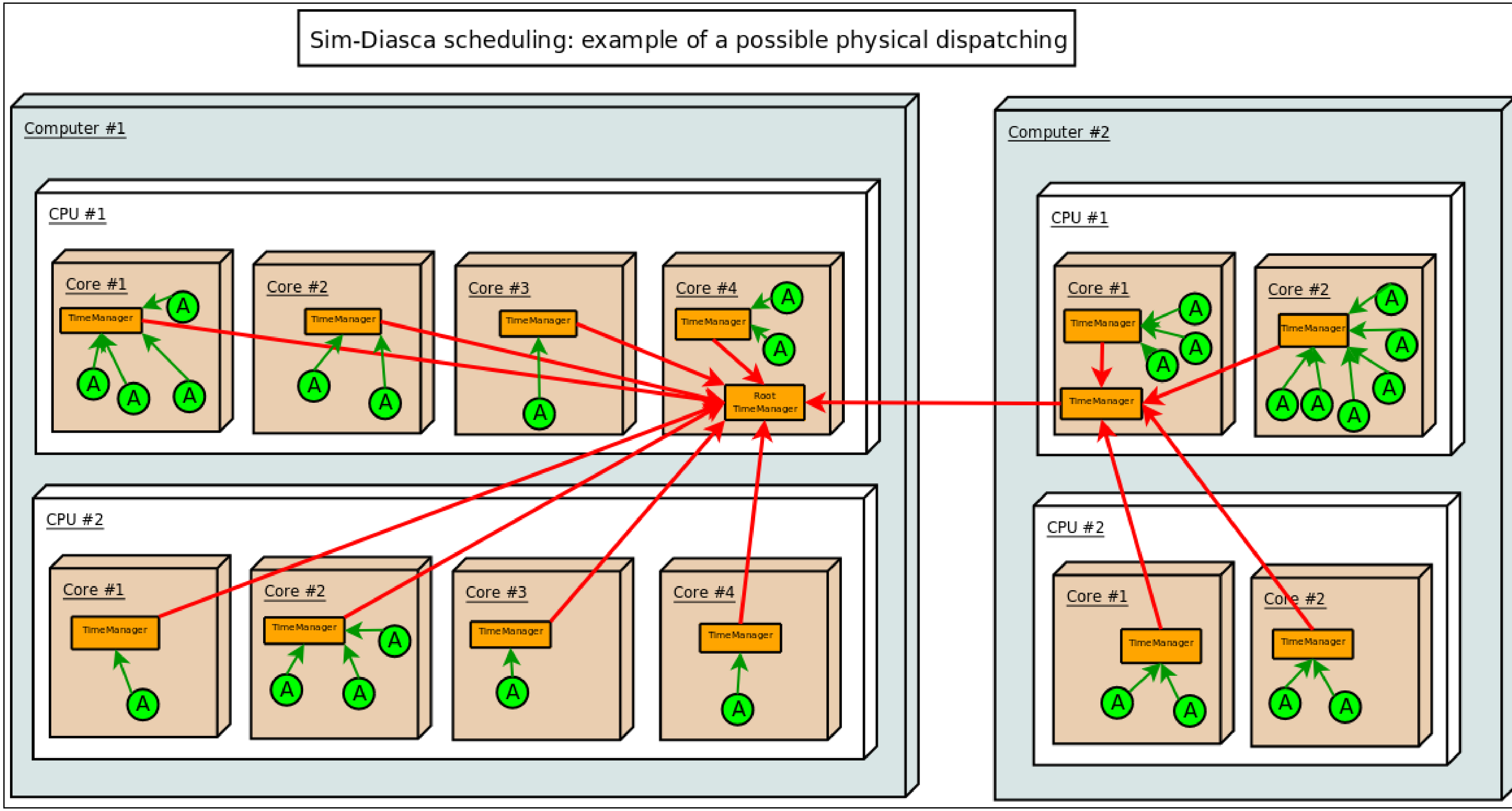
Simulation properties discussed in annex 5.

Simulation class and algorithmic aspects are better explained respectively in annex 3 and annex 6.

Overall modelling and simulation approach described in annex 7.

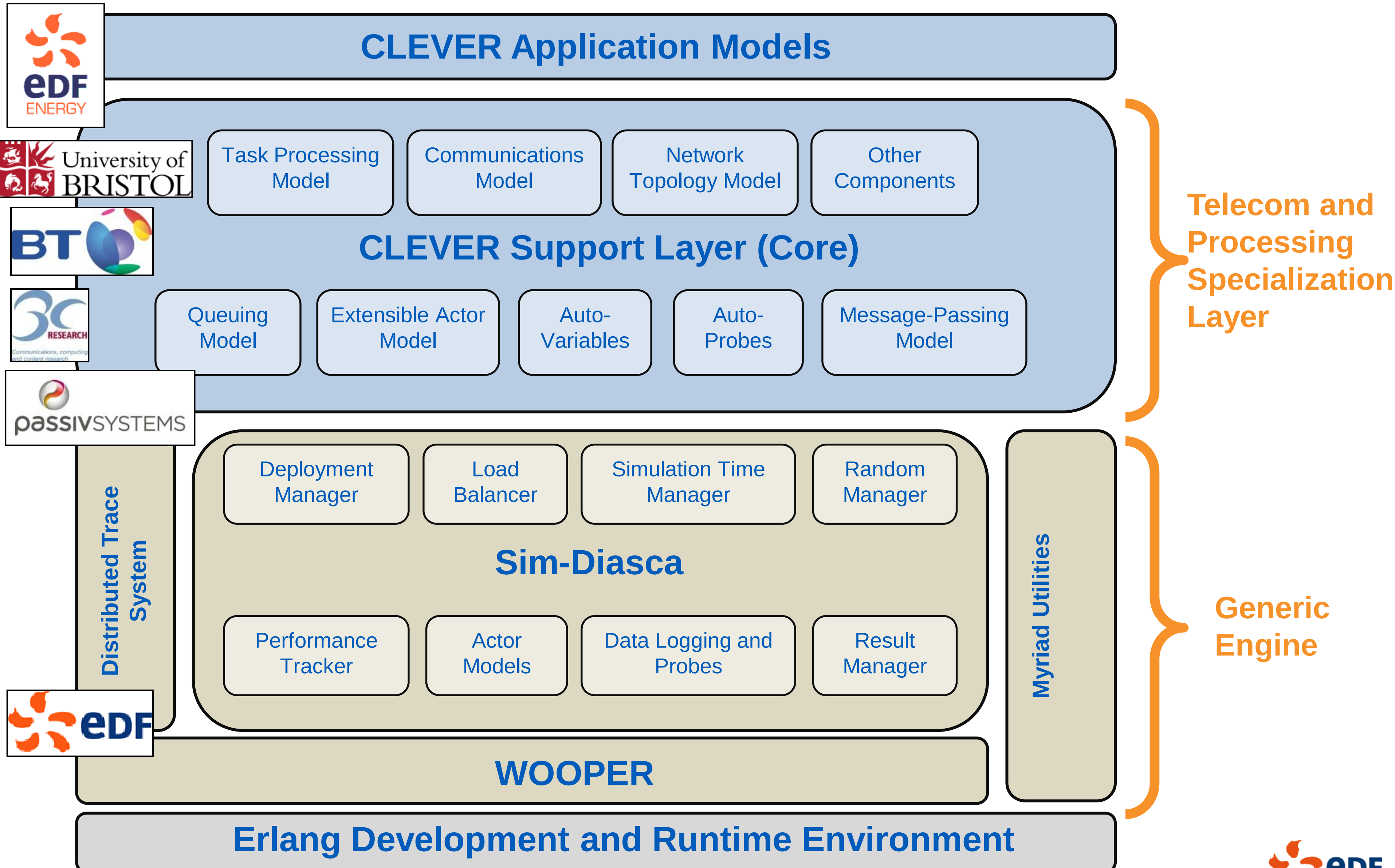
Refer to the *Sim-Diasca Technical manual* for further information.

Sim-Diasca Technical Architecture in Practice



- Most simulation services are at least partially distributed
- Dependencies on latency and bandwidth have been minimized (e.g. placement hint, advanced scheduling, hierarchical aggregation)
- High Performance Clusters supported, other platforms have been investigated (Tilera manycore cards, Bluegene/Q supercomputers)

Example of a technical architecture based on Sim-Diasca: the CLEVER simulator



eDF

Sim-Diasca: past, current and next steps

• After having been developed initially for the French case, used in the **CLEVER UK project**, in the **RELEASE european project**, in the **EDF City Plaftorm** for the **MUG Project** (not counting the external, non-EDF uses):

• Possible <u>follow-ups</u> for:

- The British supplier case and/or for OFGEM (regulator), i.e. a « CLEVER 2.0 »
- The French counterpart supplier and/or DNO (distribution operator)
- Smart grids (bridge towards equational models in continuous time, based on FMUs) and « future internet » related projects
- Projects about scalability/reliability: some other Complex Systems of interest

  (outside of the strict energy field, like urban planning, operational use of blockchains and DLT, intricated large-scale planning verification, digital twin of information systems)

• <u>On the technical side:</u>

- Plenty of improvements could be considered (e.g. half-word emulator, more metaprogramming, hibernation, native compilation, Rust binding)
- Larger-scale computing resources and k-crash resilient engine

• <u>On the theoretical/academic side:</u>

- Scalability, reliability and performances to be investigated at this level too (e.g. with Coq?)
- Towards hybrid simulations, mixing discrete time and continuous time with ODEs?
- More generally: rising interest in functional programming for the scientific field

  (e.g. EDF-CEA-INRIA 2012 Summer School)
- Even if coming from an industrial background, some opportunities of publications

Ending word

- Sim-Diasca: a fully functional simulation engine, already used in academic works and industry-related projects
- Very few scalable engines of that kind exist (most are sequential by design)
- For a better understanding of how Sim-Diasca works, read the next appendices
- Fully generic: use it to simulate your own target system!
- Various paradigms supported: multi-agent simulation, dataflow evaluation; an additional one could be the support of at least some models in continuous time (hybrid mode of operation)
- Sim-Diasca has been released as free software (LGPL) by EDF R&D since 2010
- Main requirements are fulfilled, steady progresses expected to come

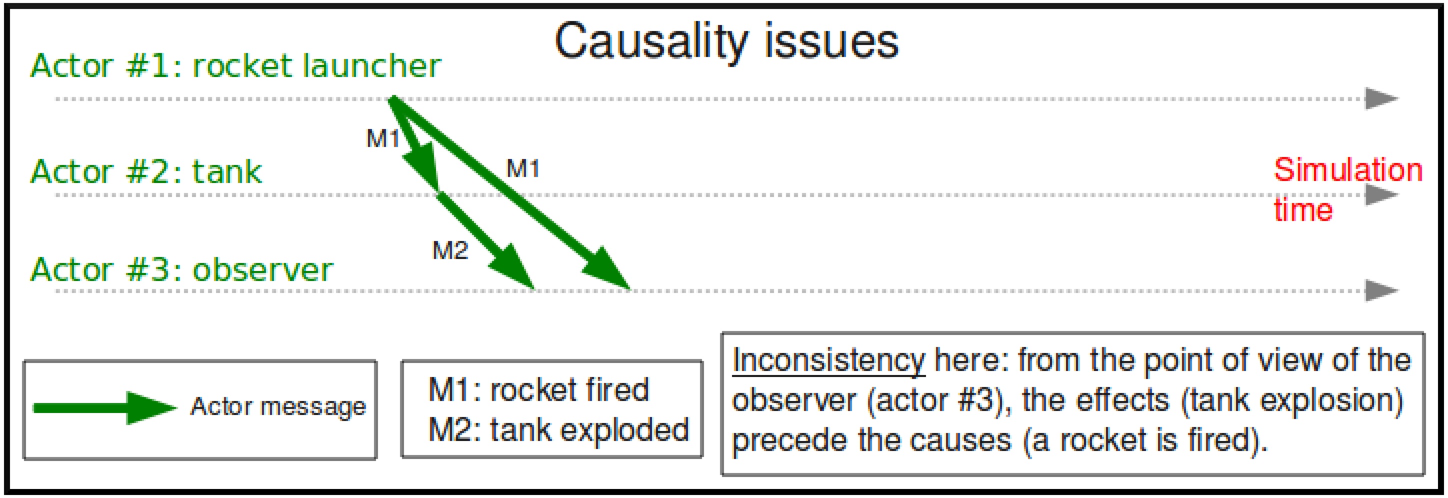
Towards a parallel, distributed, metaprogrammed Sim-Diasca running very large-scale hybrid simulations on larger HPC infrastructures?

Appendices

- Annex 1: Simulation properties
  - 1.1: Preserving causality
  - 1.2: Ensuring total reproducibility
  - 1.3: Obtaining ergodicity
- Annex 2: Mode of operation
- Annex 3: Anatomy of a Virtual Experiment
- Annex 4: Dataflow support
- Annex 5: Software-level considerations
- Annex 6: A Focus on WOOPER
- Annex 7: Overall modelling and simulation approach
- Annex 8: Examples of outputs

Annex 1.1: Obtaining targeted simulation properties: restoring causality

By default, there is no total order of events over a distributed system (as no global time can exist). If no specific order is enforced, no consistency can be guaranteed:



Causality is natively respected in Sim-Diasca thanks to a time-stepped approach:

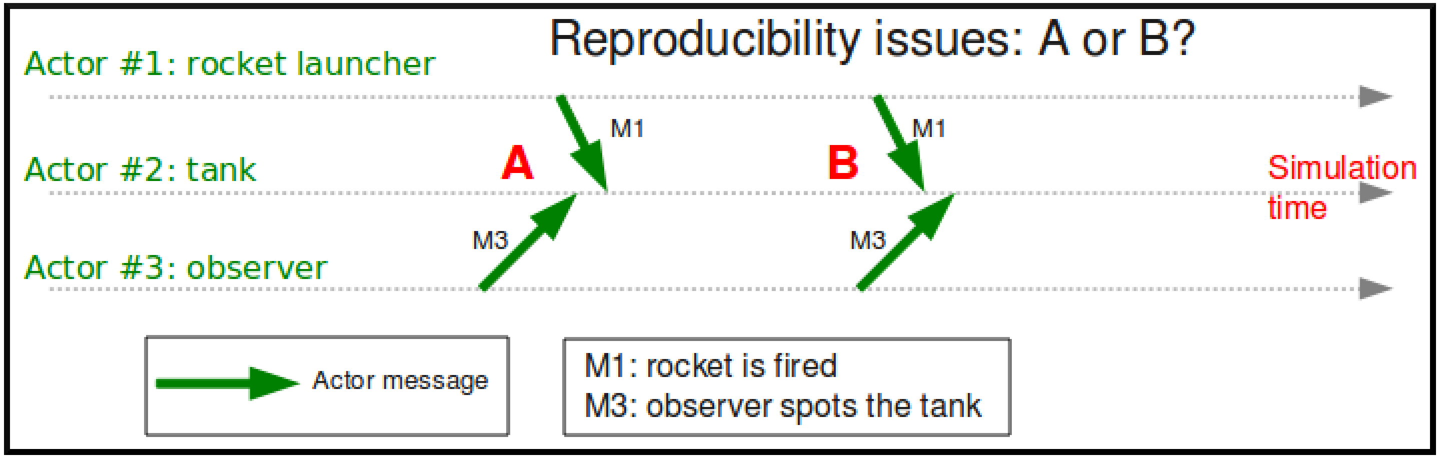
- To each cause corresponds necessarily an inter-actor message sent by actor A to actor B at tick T (*the physical time in the simulation*), diasca D (*a logical moment within a tick*)
- This message is processed (to determine its consequences) by actor B at tick T, diasca D+1

So, by design, as diascas do not overlap, causes indeed precede effects.

But causal chains induce only *partial ordering* of events.

What about *concurrent* events, i.e. events not linked by a causal relationship?

Annex 1.2: Obtaining targeted simulation properties: allowing for total reproducibility



No *a priori* order exists between two concurrent events.

(« `M1 then M3` » is not any truer than « `M3 then M1` »)

- The order that is to be re-created will necessarily be <u>arbitrary</u>.

Relying on the actual, technical order of receivings would make simulations depend on their execution context, and they would not reproducible.

- The order that is to be re-created will have to <u>fully abstract out any technical context</u>.

With Sim-Diasca, each actor is reproducibly seeded and starts a diasca by automatically reordering the messages it received on the previous diasca, based <u>only</u> on:

- a reproducible identifier of the sending actor
- a hash value of the content of the message having been sent

This arbitrary reproducible order is generic and fully compatible with parallelism & stochastic models.

Annex 1.3: Obtaining targeted simulation properties: allowing for ergodicity

Objectives:

A. Ensure at each diasca, for each actor, that:

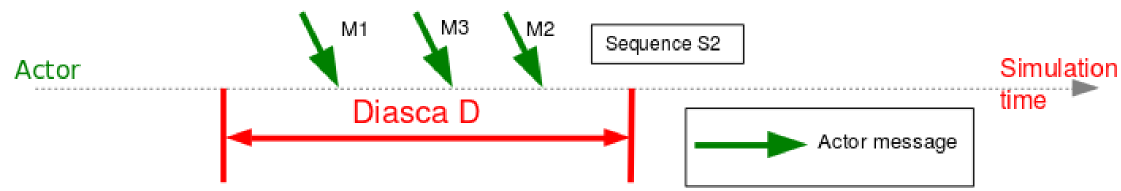
- for n messages received on the last diasca, all n! possible sequences Sk *can* occur
- And that `P(Sk)=1/n!` for k ∈ [1,n!]

B. Ensure that running a given simulation case with a given root initial random seed will fully determine the simulation outcome, for a given set of parallel and/or distributed resources involved

Sim-Diasca approach, at the level of each actor, in parallel:

- The sequence of n past messages is first arbitrarily reordered for reproducibility, as shown previously
- Then, thanks to a distributed and uniform random generator, we select (reproducibly) one of the n! possible permutations, and process it in-order
- Both objectives are then met: ergodicity (A) and reproducibility (B)

Ergodicity issues



Let's suppose that, during a given diasca D, an actor is to receive 3 messages from other actors.
Here 3! = 6 different message sequences may be considered:
- **S1**: the actor receives and processes M1 then M2 then M3
- **S2**: the actor receives and processes M1 then M3 then M2

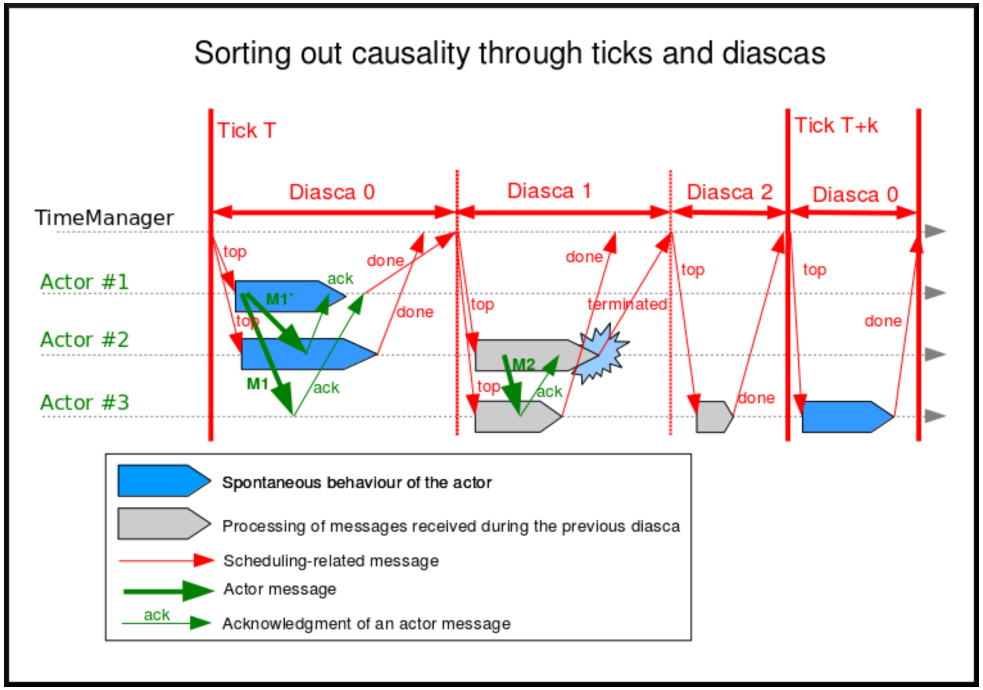
[...]
- **S6**: the actor receives and processes M3 then M2 then M1

By design all these 6 sequences should have an equal probability of showing up.

Probabilities of message sequences, as seen from the receiving actor	Based on the natural technical order of message arrival	With reproducible context-free reordering	With reproducible context-free reordering followed by an uniform permutation
S1	0%	0%	16.7%
S2	12%	0%	16.7%
S3	87%	0%	16.7%
S4	0%	100%	16.7%
S5	1%	0%	16.7%
S6	0%	0%	16.7%

Annex 2: Sim-Diasca mode of operation

Main objective: **massive parallelism while still preserving the expected simulation properties**. Should you have 35 million model instances, you are here able to evaluate all of them *in parallel*.



Note: this diagram shows how Sim-Diasca, whose simulation time is a `(Tick,Diasca)` pair, would evaluate the causality example in annex 1.1.

Following topics are not illustrated here:

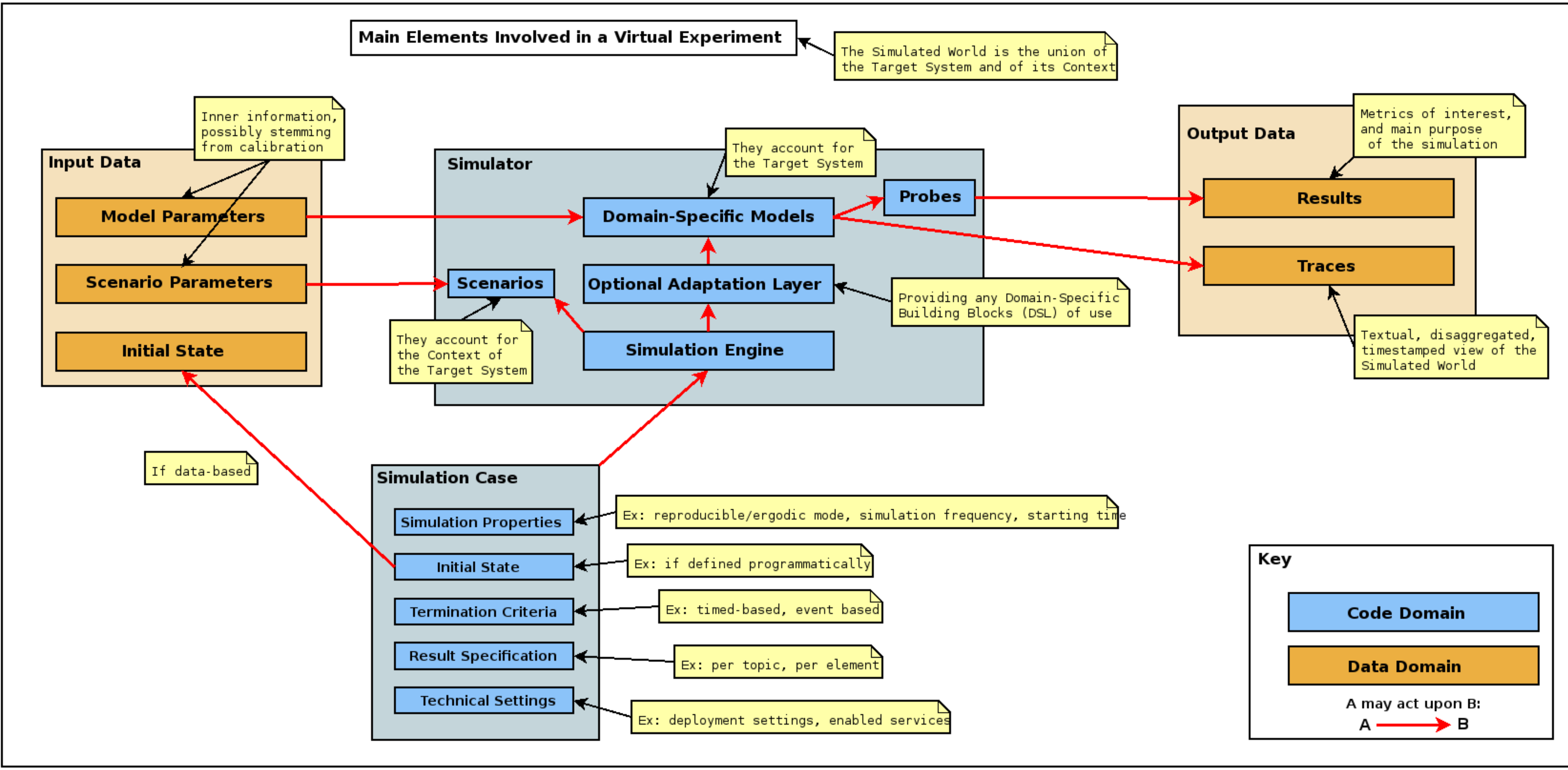
- distributed mode of operation with hierarchical time managers
- actor-level reordering of messages
- actual life-cycle management
- stochastic management

- Simulation ticks are evaluated sequentially (one after the other, based on a uniform synchronous simulation time; yet only the necessary ones), each of their diascas evaluating its actors fully in parallel

- At diasca D=0, all actors (model instances) having planned a spontaneous behaviour execute it; at D>0, those having received inter-actor message(s) at D-1 reorder and process them

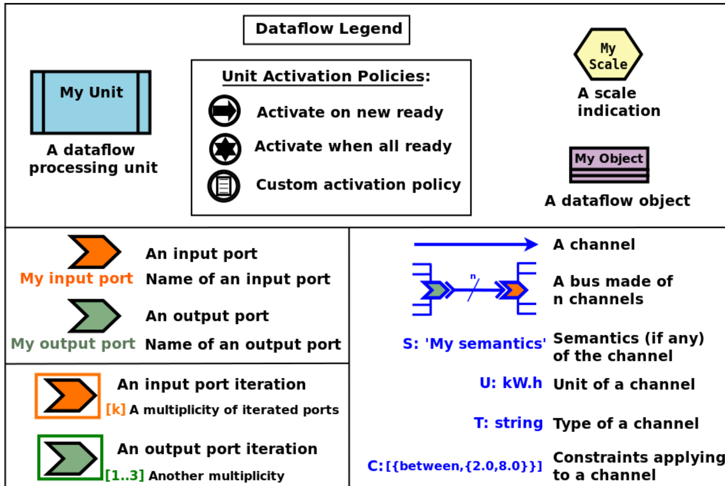
- Besides parallelism, two main goals to be achieved transparently by the engine:
  - To obtain a consensus on the correct soonest termination of each tick and diasca, thanks to a necessary and sufficient exchange of synchronisation messages (involving actors and time managers)
  - To reorder automatically received messages so that the targeted properties (causality, reproducibility, ergodicity, etc.) are preserved

Annex 3: Anatomy of a Virtual Experiment



(refer to our mini-ontology for more detailed descriptions, and to `mock-simulators/soda-test/src` for a runnable example thereof)

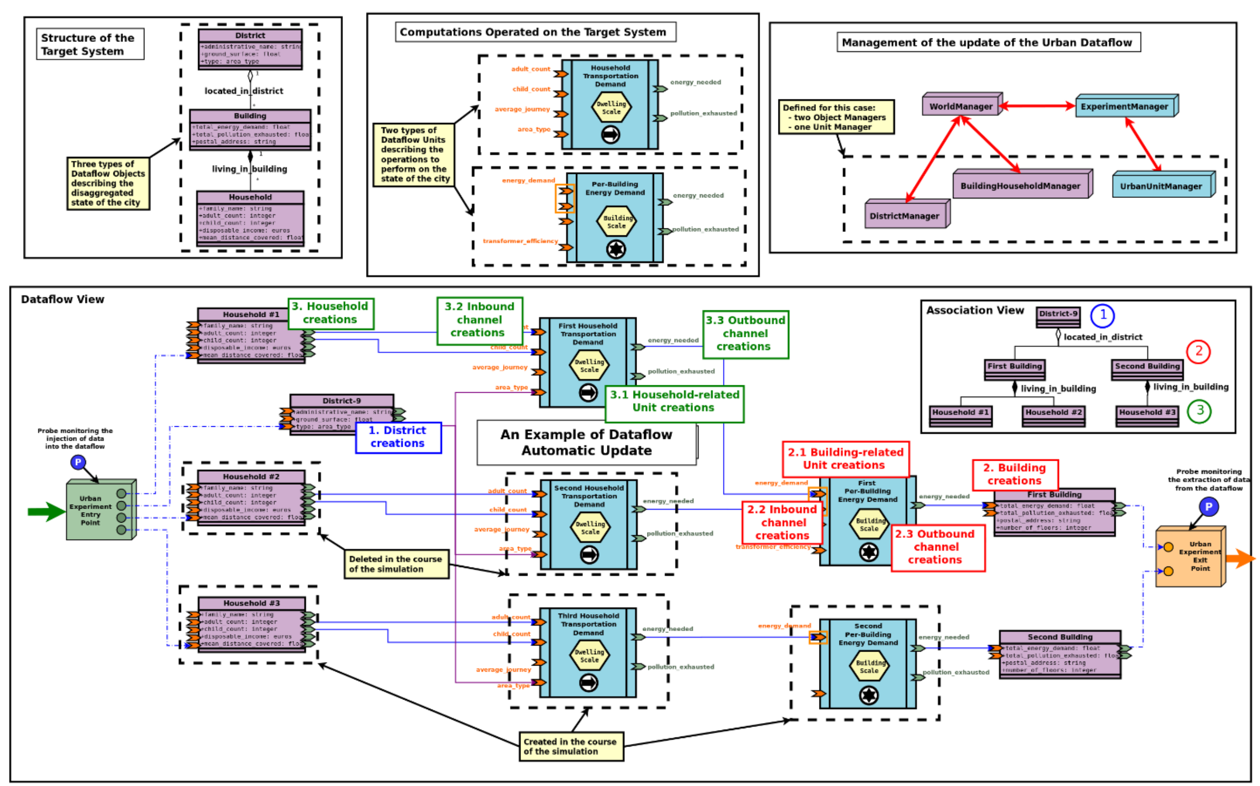
Annex 4: Dataflow support (1/2)



Sim-Diasca natively powers multi-agent simulations.

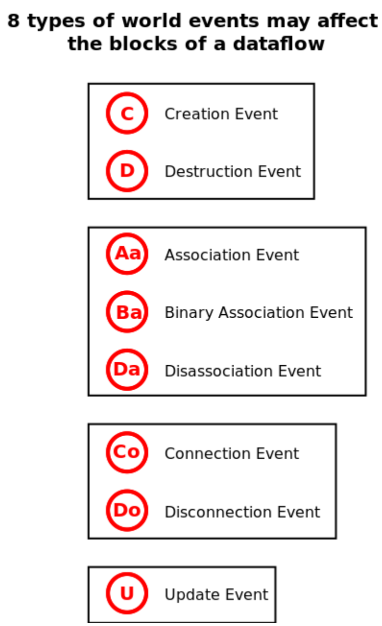
Among the specialisations that can be built on it, one deals with the parallel execution of dataflows, i.e. *graphs of computations whose evaluation is driven by the availability of inputs.*

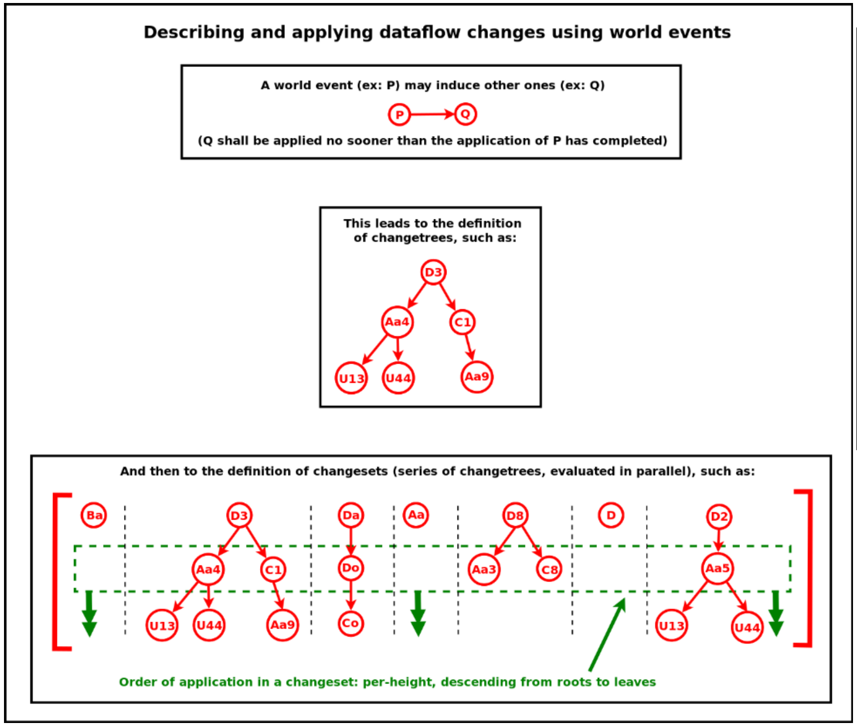
Dataflow corresponding to the Urban-Example simulation case

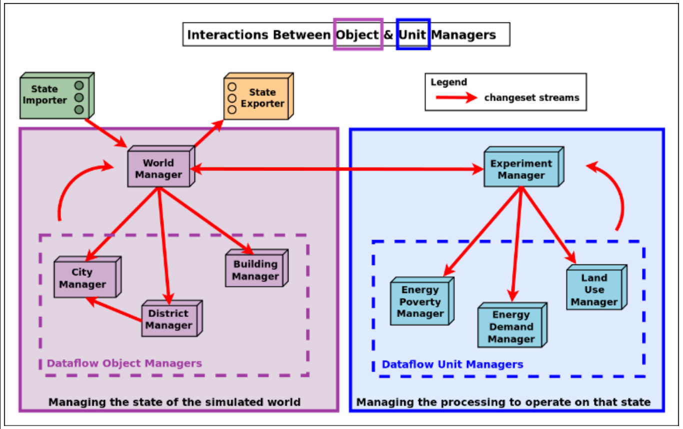


Annex 4: Dataflow support (2/2)

These dataflows can be cyclic and highly dynamic: during a time-step, any number of blocks, ports and channels can be created or destructed, based on object managers and unit managers, which exchange changesets:







Language bindings have been defined (in Python; Java considered), so that processing units may be developed in other languages and/or embed third-party pre-existing models.

A workbench has been designed to simplify such integrations.

Annex 5: Software-level considerations
A more technical zoom on the open source software stack

Through a bottom-up specialisation, obtaining in turn:

6. a **complex system** simulator	**Mock-Simulators 1.0.* (http://sim-diasca.com)**	71 .erl, 9 .hrl, 27 klines
5. a concurrent **simulation** engine	**Sim-Diasca 2.4.5 (http://sim-diasca.com)**	146 .erl, 41 .hrl, 106 klines
4. with advanced distributed **traces**	**Traces 1.0.* (http://traces.esperide.org)**	24 .erl, 11 .hrl, 16 klines
3. an **object-oriented** MAS	**WOOPER 2.0.* (http://wooper.esperide.org)**	57 .erl, 22 .hrl, 26 klines
2. **higher-level**, richer primitives	**Myriad 1.0.* (http://myriad.esperide.org)**	235 .erl, 45 .hrl, 145 klines
1. a **functional, concurrent** MAS (Multi-Agent System)	**Erlang VM (25.3 or higher, SMP)**	
	GNU/Linux x86-64 (e.g. Arch Linux)	

Grand total:
533 .erl, 128 .hrl, 320 klines
(38% of code, 31% comments, 31% blank lines)

Annex 6: A Focus on WOOPER (1/4): *Wrapper for Object-Oriented Programming in Erlang*

- A free-software lightweight OOP layer on top of the Erlang language

(now OTP-compliant thanks to rebar3, for applications/releases, and available as an Hex package)

- Official website: http://wooper.esperide.org

  (sole dependency: Myriad, see http://myriad.esperide.org)

Thus provides an object-oriented, distributed multi-agent system; but such features alone are *not* sufficient to obtain a simulation!

- Provides multiple inheritance, polymorphism, encapsulation, state management and life-cycle management with minimal development/runtime overhead

- You can define classes, create instances, call methods (oneway or requests) on them, and delete them

- A class definition includes: a list of the direct superclasses (`[class_Mammal,class_Viviparous]`), at least one constructor, possibly a destructor, and member and static methods (e.g. `declareBirthday/1`)

Annex 6: A Focus on WOOPER (2/4): implementing a class

```erlang

% @doc Cat-based example. Those are the ones that work best.
%
% Guaranteed to be implemented by a cat.
%
-module(class_Cat).


-define( class_description,
		 "Class modelling any kind of cat, and there are many." ).


% Determines what are the direct mother classes of this class (if any):
-define( superclasses, [ class_Mammal, class_ViviparousBeing ] ).


-define( class_attributes, [
	{ whisker_color, whisker_color(), none,
	  "50 shades of whiskers" } ] ).


% Allows to define WOOPER base variables and methods for that class:
-include("wooper.hrl").


% Import common types without module prefix:
-include("ecosystem_types.hrl").


% Shorthands:

-type ustring() :: text_utils:ustring().


% @doc Constructs a cat instance.
-spec construct( wooper:state(), age(), gender(), fur_color(),
				 whisker_color() ) -> wooper:state().
construct( State, Age, Gender, FurColor, WhiskerColor ) ->

	% First the direct mother classes:
	MammalState = class_Mammal:construct( State, Age, Gender, FurColor ),
	ViviparousMammalState = class_ViviparousBeing:construct( MammalState ),

	% Then the class-specific attributes:
	setAttribute( ViviparousMammalState, whisker_color, WhiskerColor ).



-spec destruct( wooper:state() ) -> wooper:state().
destruct( State ) ->

	io:format( "Deleting cat ~w! (overridden destructor)~n", [ self() ] ),

	State.



% @doc No guarantee on biological fidelity.
-spec getTeatCount( wooper:state() ) -> const_request_return( teat_count() ).
getTeatCount( State ) ->
	wooper:const_return_result( 6 ).


% @doc Cats are supposed carnivorous though.
-spec canEat( wooper:state(), food() ) -> const_request_return( boolean() ).
canEat( State, soup ) ->
	wooper:const_return_result( true );

canEat( _State, chocolate ) ->
	throw( { harmful_food_detected, chocolate } );

canEat( State, croquette ) ->
	wooper:const_return_result( true );

canEat( State, meat ) ->
	wooper:const_return_result( true );

canEat( State, _OtherFood ) ->
	wooper:const_return_result( false ).


% @doc Returns the whisker color of this cat.
-spec getWhiskerColor( wooper:state() ) -> const_request_return( color() ).
getWhiskerColor( State )->
	io:format( "getWhiskerColor/1 request called by ~w.~n", [ ?getSender() ] ),
	wooper:const_return_result( ?getAttr(whisker_color) ).


% @doc Requests this cat to terminate, based on specified halting procedure.
-spec terminate( wooper:state(), 'crash' ) -> const_oneway_return().
terminate( State, crash ) ->
	basic_utils:crash(),
	wooper:const_return().



-spec toString( wooper:state() ) -> const_request_return( ustring() ).
toString( State ) ->
	Description = text_utils:format( "cat instance with whiskers of color ~p.",
									 [ ?getAttr(whisker_color) ] ),
	wooper:const_return_result( Description ).



% Static section.


% @doc Returns the mean life expectancy of a cat, in years.
-spec get_mean_life_expectancy() -> static_return( age() ).
get_mean_life_expectancy() ->
	wooper:return_static( 18 ).
```

Annex 6: A Focus on WOOPER (3/4): interacting with instances

```
MyC = class_Cat:new_link( _Age=3, _Gender=female, _FurColor=sand, _WhiskerColor=white ),

MyC ! { getClassname, [], self() },
receive
	{ wooper_result, class_Cat } -> ok
end,

MyC ! { getSuperclasses, [], self() },
receive
	{ wooper_result, _Classes=[ class_Mammal, class_ViviparousBeing ] } -> ok
end,

MyC ! { getAge, [], self() },
receive
	{ wooper_result, 3 } -> ok
end,

MyC ! { setAge, 5 },

MyC ! { getAge, [], self() },
receive
	{ wooper_result, 5 } -> ok
end,

MyC ! declareBirthday,

MyC ! { getAge, [], self() },
receive
	{ wooper_result, 6 } -> ok
end,

MyC ! { canEat, soup, self() },
receive
	{ wooper_result, true } -> ok
end,

MyC ! delete,

18 = class_Cat:get_mean_life_expectancy().
```

Annex 6: A Focus on WOOPER (4/4): Inner workings & Erlang mapping

- A WOOPER class is an (Erlang) module (e.g. `class_Cat`), a WOOPER (active) instance is an (Erlang) process, an instance identifier is a PID, method calls are messages, a state is a set of attributes, an attribute is a key/value pair (`{atom(),term()}`)

- A WOOPER instance is:
  - created thanks to `(timed_)(remote_)(synchronous_)new(_link)` calls
  - a process looping over its state, waiting for incoming method calls, mapping them to the corresponding functions in appropriate modules (inheritance), returning possibly a result
- It keeps a private associative table to hold its state (`attribute_name -> value`)

WOOPER concept	Corresponding Erlang mapping
class definition	module
instance	process
instance reference	process identifier (PID)
new operators	WOOPER-provided functions, making use of user-defined `construct/N` functions (a.k.a. the constructors)
delete operators	WOOPER-provided functions, unless user-specified (a.k.a. the destructor)
method definition	module function that respects some conventions
method invocation	sending of an appropriate inter-process message
method look-up	class-specific virtual table taking into account inheritance transparently
instance state	instance-specific datastructure, kept by the instance-specific WOOPER tail-recursive infinite loop
instance attributes	key/value pairs stored in the instance state
class (static) method	exported module function

Annex 7: Overall modelling and simulation approach (example from the CLEVER project)

Massive roll out of smart meters in Great-Britain

A challenge raising questions

Volume of data transfers | Associated costs | Scalability of the system

What is the most appropriate smart metering architecture?

DEFINE a Set of Experiments

Covering all questions raised by a smart metering systems roll-out

DESCRIBE

Business processes

for Smart Metering from meter reading to advanced DSM including future needs

CHARACTERISE

Architecture options

System structure, telecom solutions

DEVELOP and VALIDATE

Simulation engine & tools

Simulator, models, computing resources, formal descriptions

RUN the Set of Experiments

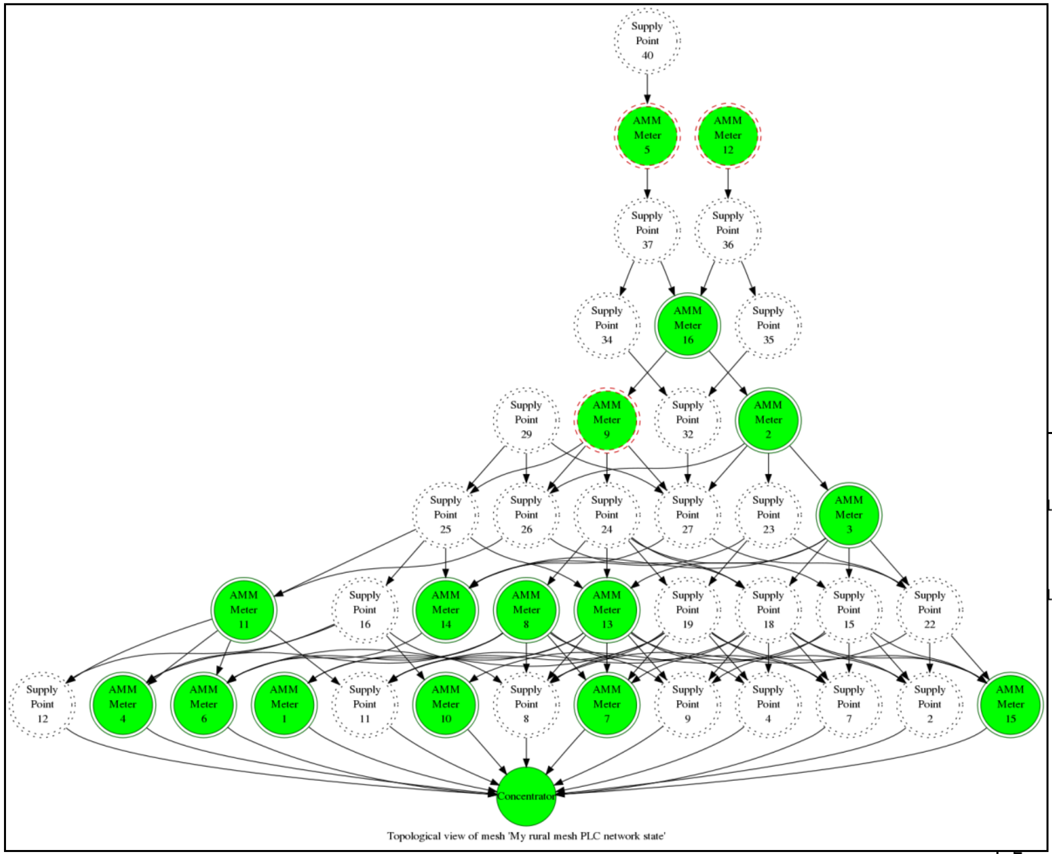
To get raw results: curves, probe indicators

ANALYSE Results

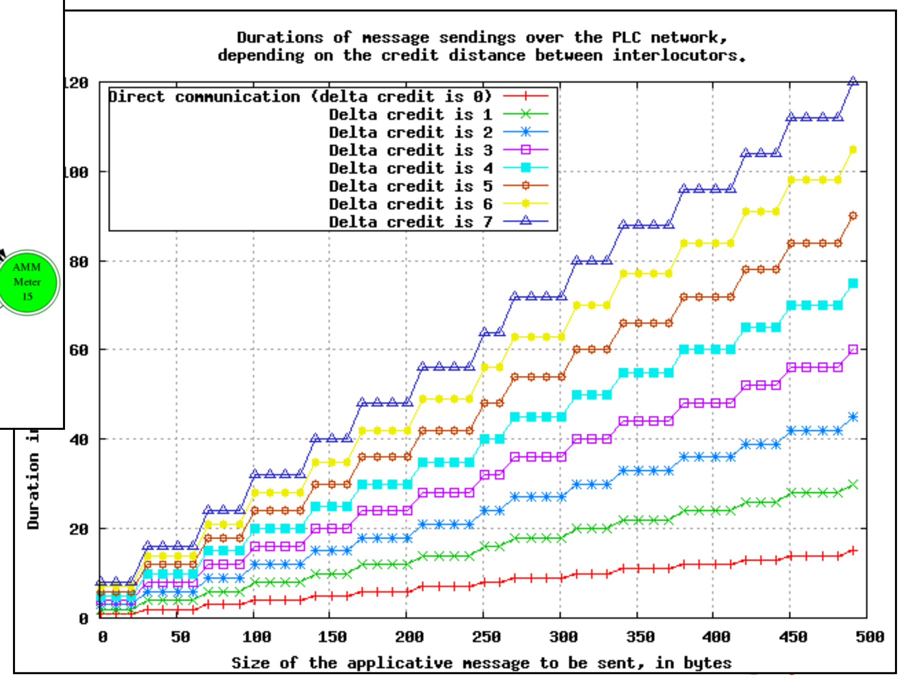
To interpret data and formulate clearly understandable conclusions

Annex 8: Examples of Sim-Diasca results & outputs (1/3)

Simulation of meters communicating with a concentrator through PLC G1 (French case, 2008)

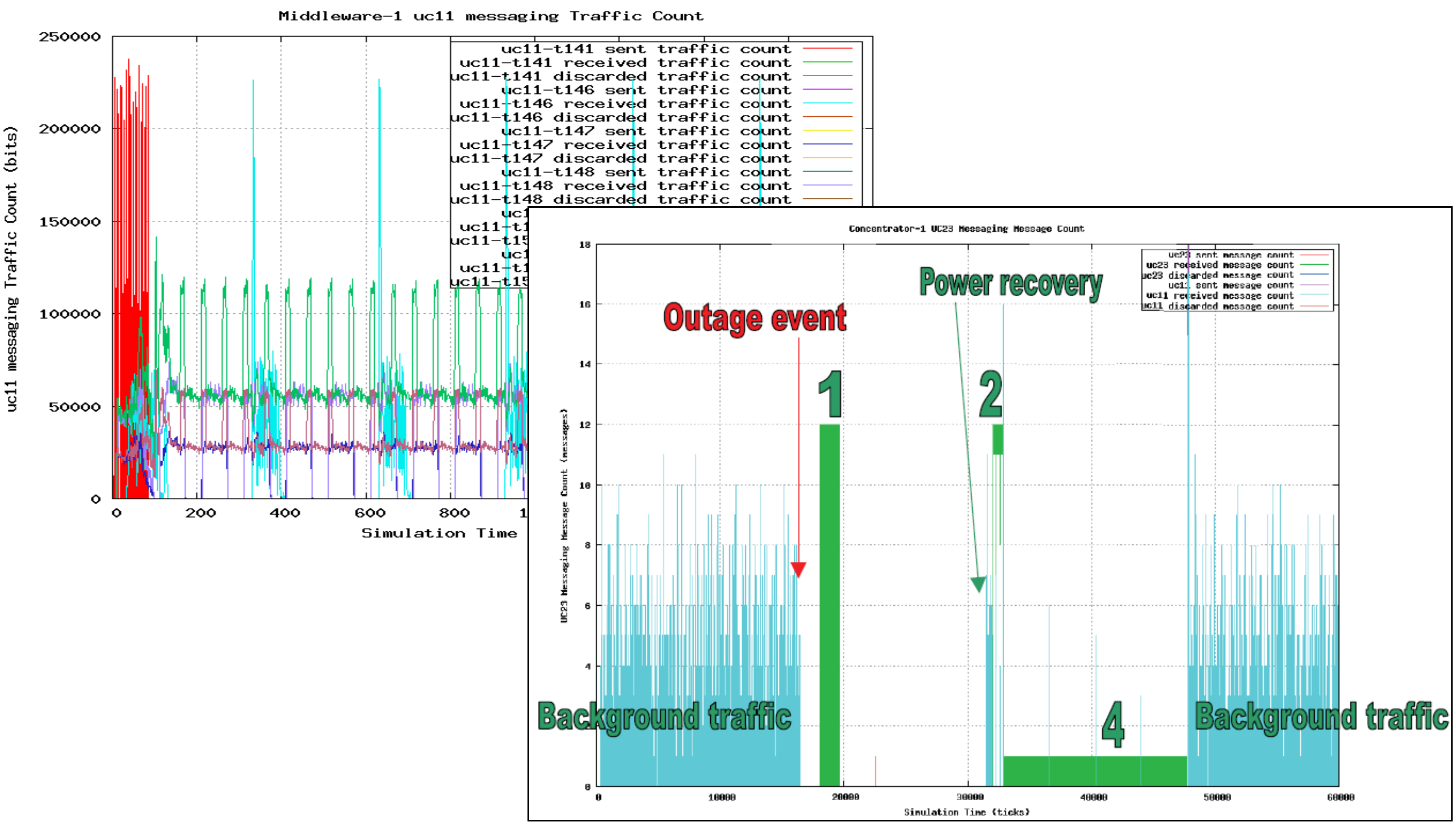


Topological view of mesh 'My rural mesh PLC network state'



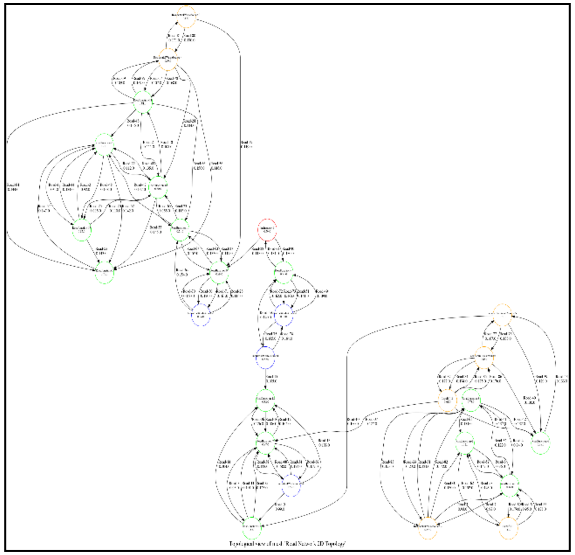
Annex 8: Examples of Sim-Diasca results & outputs (2/3)

Simulation results obtained through the CLEVER simulator: evaluating outcomes and performances depending on the functional and technical architecture of various smart metering systems (British case, 2009-2011)

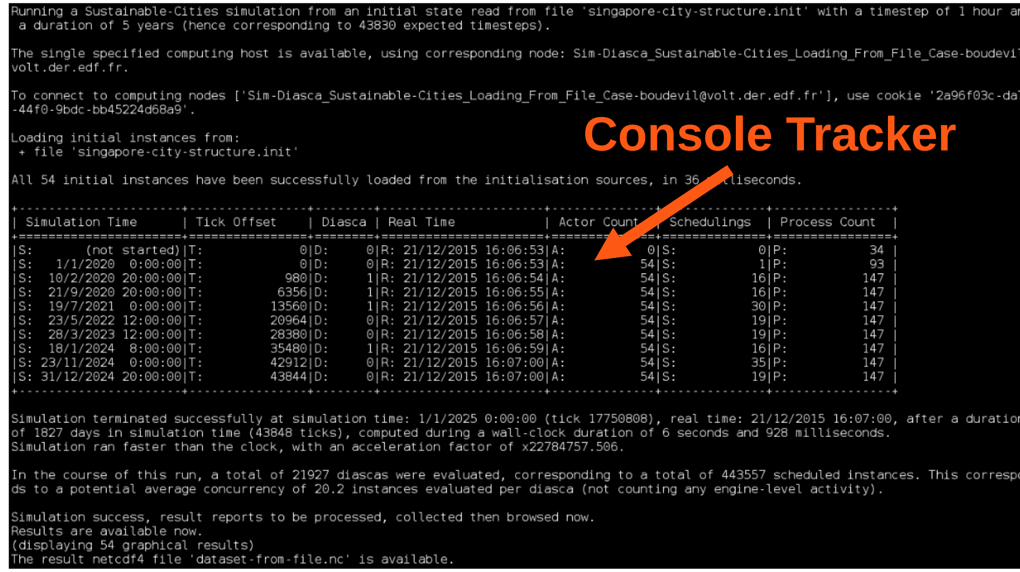


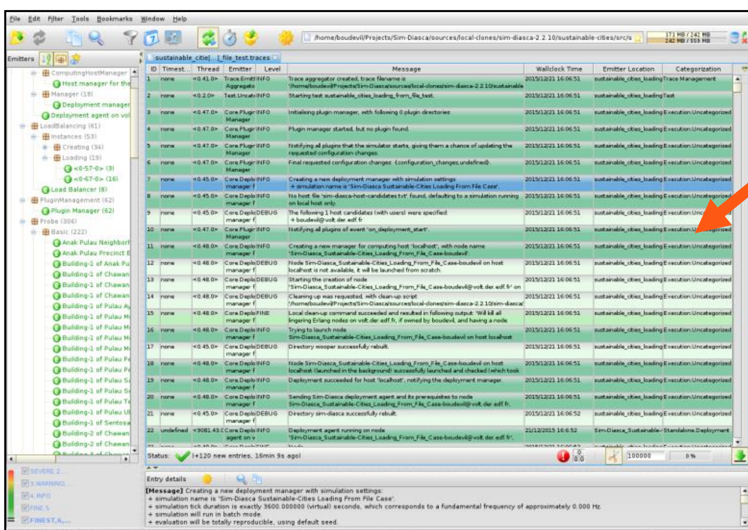
Annex 8: Examples of Sim-Diasca results & outputs (3/3)

Simulation results obtained, on the left, through the RELEASE European project (large scale City-example case, here the road network of the smallest scale); on the right, internal mock-up Sustainable Cities case, devised for platform integration.



Road network (for scale 'tiny')





Trace Browser