MetroC: A METROPOLIS BASED DESIGN METHODOLOGY DEVELOPED IN A C++ FRAMEWORK

by

Daniele Gasperini, Alessandro Pinto and
Alberto Sangiovanni Vincentelli

Memorandum No. UCB/ERL M05/6

1 January 2005

MetroC: A METROPOLIS BASED DESIGN METHODOLOGY DEVELOPED IN A C++ FRAMEWORK

by

Daniele Gasperini, Alessandro Pinto and
Alberto Sangiovanni Vincentelli

Memorandum No. UCB/ERL M05/6

1 January 2005

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

MetroC: a metropolis based design methodology developed in a C++ framework

Daniele Gasperini[1], Alessandro Pinto[1], Alberto Sangiovanni Vincentelli[1]

University of California at Berkeley, Berkeley CA 94720, USA

Abstract. This overview describes in general terms a `MetroC` library and how it simulates. The `MetroC` library of classes and simulation kernel extend C++ to enable the modeling of systems. The `MetroC` library is based on three components: a metropolis metamodel simulator (core-lib), an xml based type exchange format (types-lib) and a co-simulation library (hw_cosim). The library is developed in C++ thus allowing the following benefits: speed, portability (it uses the Posix standard), and easy integration with existing tools (i.e. modelsim, systemc, matlab, instruction set simulators, etc ...). Indeed the three components can be organized in such a way that our Metropolis metamodel simulator is able to co-simulate with another instance of itself thus providing a distributed, highly scalable and efficient architecture. The framework presented here supports the platform based design methodology.

1 Introduction

The key elements of the library are depicted in the picture below along with their relationships:

- core-lib: a C++ library and simulation kernel that extends C++ using the semantics of the metropolis metamodel
- types-lib: an xml based encapsulation mechanism to allow interchange among different domains of data according to well-established interfaces
- hw_cosim: a C++ library based on the adapter design pattern (see 7.1) that provides the necessary layer that allows two different domains communicate and simulate according to a pre-defined schema. This schema is implemented in the adapter. Since the communication among different domains is an ill problem the implementation has to provide some rules to resolve it.

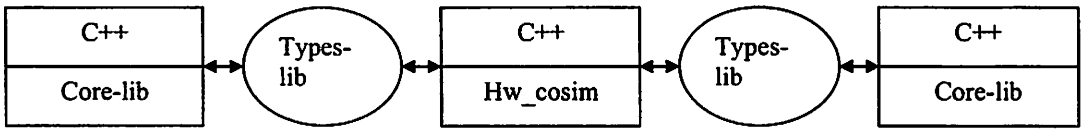


Fig. 1. Library components

1.1 Metropolis metamodel

The core library uses the semantics of the metropolis metamodel. The core of the infrastructure is a meta model of computation, which allows one to model various communication and computation semantics in a uniform way. By defining different communication primitives and different ways of resolving concurrency, the user can, in effect, specify different models of computation (MOCs). At a high level of abstraction, the designer may want to use an MOC that is convenient to describe the functionality.

The functional model just represents what a system is supposed to do. On the other hand, metropolis allows the description of architectures and finally the mapping of the former to the latter. The result of a mapping is another model whose level of abstraction is lower that the original functional model. Refer to figure 2.

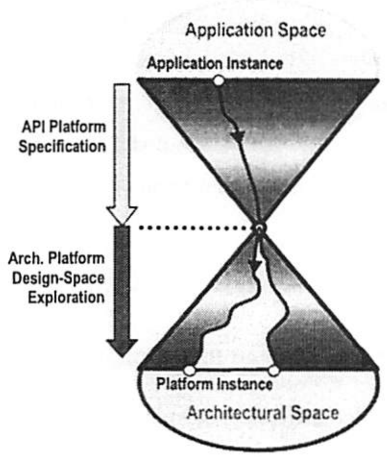


Fig. 2. Platform based design

The word platform is used to refer to three different kind of platforms:

- the hardware platform, a family of architectures that satisfy a set of architectural constraints that are imposed to allow the re-use of hardware and software components;
- the software platform, a layer that wraps the essential parts of the hardware platform in order to allow application software "sees" a high level interface to the hardware (the Application Program Interface or API);
- the system platform, the combination of hardware and software platform.

In figure 2 the term platform-stack is used to refer to the combination of two platforms (software and hardware) and the tools that map one abstraction into the other. The platform-stack can be seen as a "single" layer obtained by gluing together the top platform and the bottom platform whereby the upper view is the API platform and the lower view is the collection of components that comprise the architecture platform.

When the upper part is mapped on the bottom part, a system platform instance is chosen for that particular stack at a certain level of abstraction. In order to abstract the definition of the hardware platform the use of different models of computation are allowed at different abstraction levels. There are many models of computation which differ in computing power (that is, some models can perform computations impossible for other models) and the cost of various operations. Metropolis provides a unified framework that allows a system to be represented at both of these abstraction levels, as well as in any combination of the two that arises as parts of the design are refined and implemented. The meta model is used to represent the function of a system being designed at various levels of abstraction, represent the architectural targets, allow mapping between different platforms, to generate executables for simulation, and as input to formal methods built in Metropolis for both synthesis and verification in various design stages.

The goal of the `metroc` core library is to provide the necessary infrastructure to build software and hardware platforms in such a way that each platform stack can be easily explored and connected with other platform stacks.

The key articulation points around which the metropolis framework is built are:

- the basic components that glued together give birth to the final system (netlists, mediums, processes, ports, etc);
- a very light, generic and extendible simulation core capable of accommodating different models of computation (MOC). Each MOC may be developed as a library extension on top of the core library. An example of such extension is the xGiotto library.
- a generic co-simulation framework that allows different platform stacks talk together.

The way the different platform stacks are developed depends on the abstraction level: at top levels C++ code based on Metropolis framework may be more suitable, whereas at the bottom levels VHDL/Prolog models of hardware IP blocks may be used. In order to provide an unified framework that allows different stacks exchange data and use services (API) two libraries have been developed:

- a co-simulation library that abstract the services by providing a single common API;
- a library of basic types to handle generic data exchange.

The type library is developed as a set of C++ classes with the same interfaces as C++ basic types. The common operations have been overloaded (plus, minus, etc...) thus using C++ basic types or the corresponding type library classes is exactly the same. The power of the type library classes is exploited during data exchange among different and enterogenous domains where they offer a marshalling mechanism (see section 3) to the co-simulation library.

The methodology is based on defining Platforms at the different key articulation points in the design

flow. Each platform represents a layer in the design flow for which the underlying, subsequent design-flow steps are abstracted.
The proposed design flow is shown in figure 3.

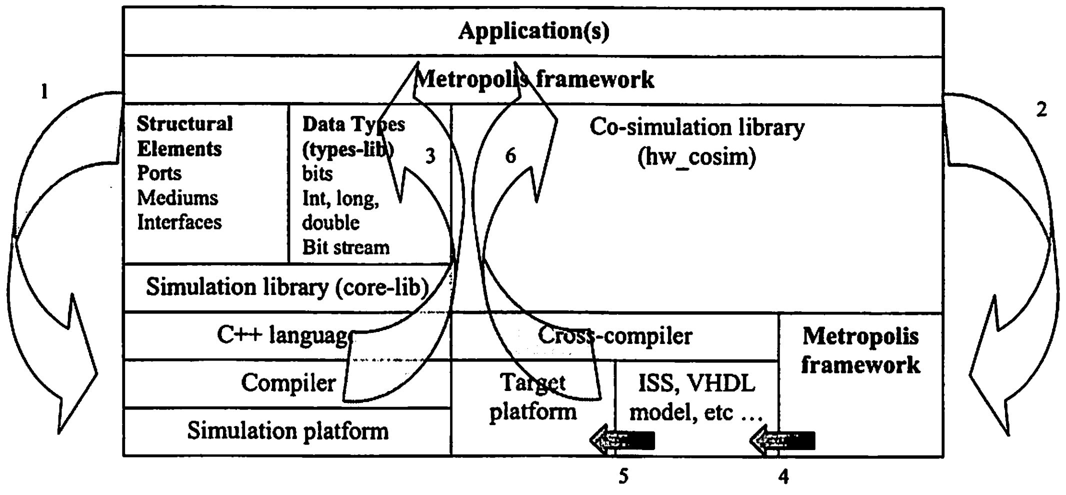


Fig. 3. Design flow

At the top level of abstraction the main functionalities are captured throughout different processes. Once the functionalities are established, these functionalities are tested on a simulation platform (step 1). This platform is purely functional and does not correspond to the target platform. It may have a completely different architecture in terms of cpu, memory, storage, etc. This design process is iterated (step 1-2) until the overall functional model is working.
The design flow of the functional model follows the standard software design cycle. Software life cycle deals with the all activities and work products necessary to develop a software system.
Four fundamental process activities are usually necessary

1. software specification (requirements, functionality and constraints)
2. software development (design and implementation)
3. software validation (ensure that the software meets the customer needs)
4. software evolution (evolve to meet changing customer needs)

The idea of platform based design is to extend software design patterns to hardware platforms moving from left side of figure 3 to right side of figure 3.
Each platform stack can be seen as merging together a chosen software platform instance with a chosen hardware platform instance. According to the level of abstraction step 2 may be realized in the same semantic domain of Metropolis framework or in another domain. Consider for example the task of developing an embedded system, from the low level RTOS up to the applications. The API provided

by the OS may be developed in metropolis itself. When the platform stack (step 4,5 and 6) considered is the one that glues together the RTOS and the hardware components of the embedded system it may be more suitable a description in VHDL or verilog of the very same IP blocks.

In order to mingle together different heterogenous subsystems a high level interface along with a predefined set of API to access it have been defined. Each component (either software or hardware) provides a set of services throughout this interface. In the implementation this interface is an unique and generic function called `execute()`.

This `execute()` is the connecting point between the upper part of the ASV triangle and the lower part. For example, in the Linux OS the `execute()` function role is carried out by `int 0x80`. In a cpu model the call to `execute()` means fetching from memory one or more instructions, execute them and return the results. At all level of abstraction one can find that this paradigm is true. Since the interfaces are different (`int 0x80` in Linux, fetch, decode and execute in a CPU model) according to the type of services offered, the only possibility is to abstract the concept of service. A service is like a `black box` with inputs and outputs. Once the inputs and outputs are standardized and the internal behavior of the `black box` is known (the implementor of the black box has captured a feasible behavior to resolve the co-simulation problem) then the user of the `black box` may just call the `execute()` function that a feasible execution trace is ensured by construction.

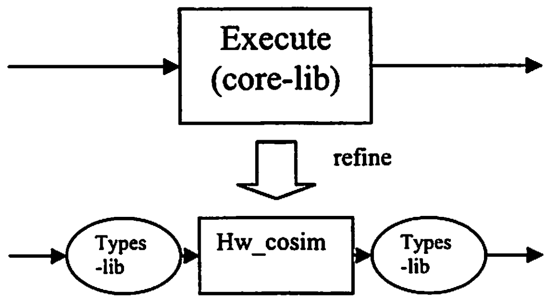


Fig. 4. The execute blackbox

The `execute` function role is carried out by the `atom` in the core-lib. An `atom` abstracts all the events between the begin and end event of the `atom`. For instance, an `atom` cannot contain a request to a quantity because the events in an `atom` are not visible. An `atom` is the smallest execution unit that is executed without interruption (by the simulator core) from the begin up to the end. In core-lib compositions of atoms are allowed. One very natural composition is the sequential one that, if things are well architected, will make it possible to map Metamodel processes to `Metroc` processes. Another one is the parallel composition, that will allow many Metamodel processes to be mapped to one `Metroc` process.

An atom can be refined in the same core-lib domain or mapped to another domain. Another domain may be an external tool (i.e. an instruction set simulator, a VHDL simulator, etc ...) or another core-lib based block (i.e. a network of nodes developed in Metroc that communicate among them, a discrete domain and a continuous domain, etc ...).

This atom in this exception is commonly called IP block and can be (arrow 3):

1. written in another language/ external tool, hence with a different execution semantics;
2. written in metropolis itself.

In the first case there are two possibilities:

1. use an adapter specific for the language/external tool already present in the library (at the moment VHDL modelsim and ISS simplescalar);
2. develop its own adapter for its specific language/external tool.

In case the IP block is written in Metropolis, two scenarios are possible:

1. map the functional part onto the architecture directly;
2. co-simulate the functional part with the architecture.

It is always a mapping but the difference is the simulation and coordination strategy.

In all the cases listed above, before the user starts step 3, he can think about different platforms without the problem of the implementation, since all the pieces are like LEGO bricks that match each other in many different scenarios.

Let us call them plaforms. "A hardware platform is a family of architectures satisfying a set of constraints imposed to allow reuse of hardware and software components" (A.Sangiovanni Vincentelli).

In the figure below the Metroc framework is compared with ASV triangles ().

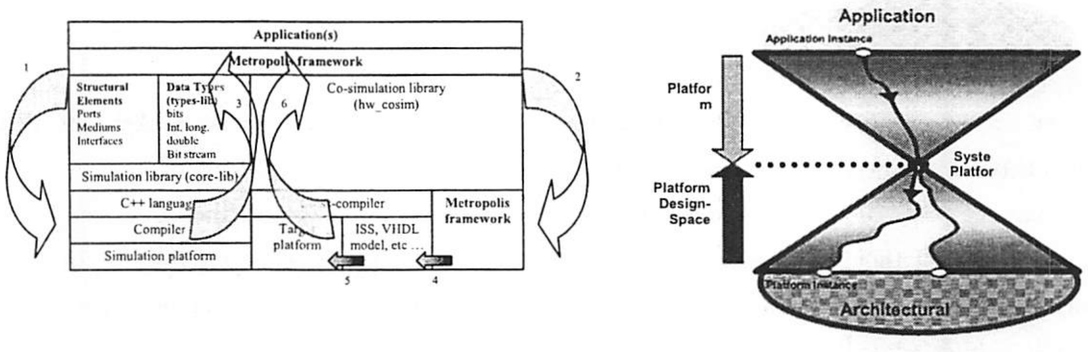


Fig. 5. Design flow

By rotating of 90 degree the figure on the left it is possible to observe the number of similarities: ASV triangles are the idea and `Metroc` framework a feasible implementation.
An important key element is how well the hardware platforms can be handled like software platforms. Hardware platforms has to be "extended" upwards to be really effective in time-to-market (ASV).
As already stated, in order to achieve this aim a number of API are usually used. Every domain specific layer has developed its own API: the RTOS try to adhere to Posix standard, CAN buses to CAN standard, Intel Pentium family to x86 standard, etc ...
The hardware co-simulation library provides a way to abstract from domain specific interfaces by providing a unified view. The implementation underneath (called adapter) must, of course, be aware of the particular interface but not the outside user.
How a particular `atom` is mapped onto a particular component of a chosen platform is up to the mapping mechanism. The granularity of the `atom` is up to the system designer. In particular an `atom` may correspond to a single operation, a statement, a bunch of statements or an entire algorithm (i.e. mp3 decoder). An `atom` in the framework is always executed between the begin of the corresponding event and its end. The way in which the events are handled is a matter of the MOC used and it is specified directly by the user.
By using this general concept an `atom` in the functional part is said to be mapped onto a corresponding architecture component by synchronization constraints on the events.
These synchronization constraints allow the user change mappings dynamically to test different configurations.
The `atoms` on the hardware part, according to the level of abstraction and the corresponding refinement level, may correspond to:

1. other `atoms` developed in the Metropolis metamodel;
2. piece of assembly code that runs on an instruction set simulator;
3. piece of assembly code that runs on VHDL/Verilog model.

Since the interfaces are well established the overall refinement process is guaranteed to be correct.
Therefore after step 3 the user may:

- develop a high level component for a particular purpose (in metropolis for i.e. but any other language such as systemC is allowed too as long as an adapter is provided in the co-simulation library);
- pick an existing IP block and try to map some actions on it;
- develop its own IP block in its favorite environment.

These steps may not be performed in a pre-defined manner. The user may prefer re-use of existing IP blocks.
As for the software part in the hardware part a sort of development cycle takes place. After step 3 the

user maps some actions on the hardware until he reaches a satisfactory compromise between software and hardware (it depends strongly on the application and no assumption is made here).
For each step 3-4-5 part of the actions described in software are cross-compiled on the target system.

2 Metropolis Simulator

2.1 The mm_process

In Metropolis a process is an active element with its own control thread.
The declaration of a process involves four steps:

1. a name used to identify the process
2. a process header (enum_static_events) where the list of actions that a process can perform are listed in form of a pre-defined list of events (an action is executed between the begin and end of an event)
3. the main thread that is the entry point of the process
4. the actions along with the statements that belong to each action

When the process hits a synchronization point (label annotation, await statement, action statement) the simulator core-lib schedules the process according to the following figure (figure 6)

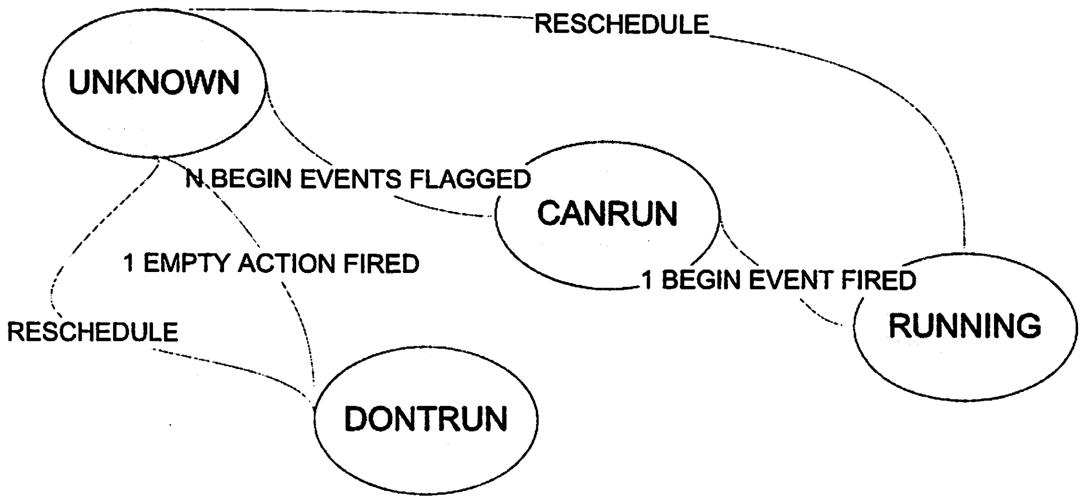
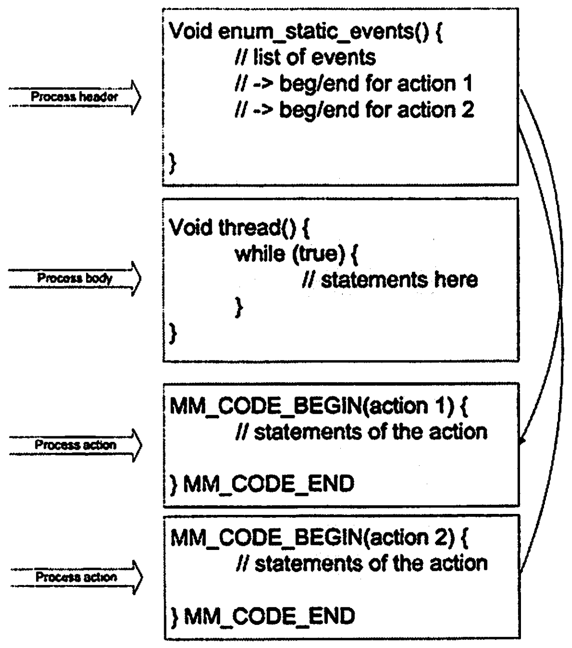


Fig. 6. Process state diagram

The process state chart follows the definition in the Tagged Signal Model. Each process has a set of behaviors. A system is determinate if it exposes exactly one (RUNNING) or zero (DONTRUN) behaviors. The way in which the set of possible behaviors (CANRUN) are restricted is determined by input signals, which are collections of events. It is up to the fire_mgr (2.5) in the core-lib intersects the events and selects the behavior.

A process skeleton is depicted in figure 7.



```
Void enum_static_events() {
        // list of events
        // -> beg/end for action 1
        // -> beg/end for action 2

}

Void thread() {
        while (true) {
                // statements here
        }
}

MM_CODE_BEGIN(action 1) {
        // statements of the action

} MM_CODE_END

MM_CODE_BEGIN(action 2) {
        // statements of the action

} MM_CODE_END
```

Fig. 7. Process skeleton

The name of the process is defined by the user and it is useful during the mapping process. The so-called process header enum_static_events is a pure virtual function that requires user implementation. The following snippet of code shows an example:

```
void myprocessclass::enum_static_events() {
    // if the process does not derive directly from mm_process
    // call the base class ! (i.e. xGiotto)
    baseclass::enum_static_events();
    // list of events/actions
    add_static_event(eventname, &actionfunction);
}
```

where `eventname` is a string with the name of the event and `actionfunction` is the address of the action to be executed when the event is fired. The `actionfunction` may also be a NULL pointer. There are at least two cases in which this may be useful:

1. the action is mapped on a co-simulation engine
2. the action represents a quantity constraint (time delay, etc ...)

In the former case all the statements belonging to the action are actually executed in the co-simulation engine. In the later case the quantity annotation requests the corresponding quantity manager to schedule the execution and the statements of the action follow the request without the need of a stand-alone action (conceptually are the same but the simulator allows both representations for a sake of simplicity from user perspective).
The following code shows a simple example for the first case:

```
...
// call the co-simulation for decoding a stream
mm_action decode(MM_COMPOSITION_SERIAL, 1, new mm_code("decode", NULL, this, "decoder.xml"));
decode.execute();
```

For the quantity case let's imagine to model a RTOS. A `taskSwitch` action may be defined as:

```
void RTOS::enum_static_events() {
    add_static_event("taskSwitch", NULL);
    add_static_event("taskRun", &taskRun);
    add_static_event("checkWaitingProcesses", &checkWaitingProcesses);
}
```

If we want to model the fact that it takes one cycle:

```
mm_action action(get_action("taskSwitch", this));
xi = (mt_unsigned_long_t *)get_quantity("_GTIME_")->A();
mm_event *event = get_static_event(MM_BEGIN_EVENT, "taskSwitch");
get_quantity("_GTIME_")->get_mgr()->request(new mm_request_class_gtime(event, xi));
get_quantity("_GTIME_")->get_mgr()->request(new mm_request_class_gtime(event->get_other(), new m
action.execute();
```

The code above does the following:

1. create the action `taskSwitch` by referring to its static event (the static event is no more static since it is created as a new instance, copy of the static one, dynamically fire-able by the simulation platform)
2. the quantity `GTIME` is called to get the current value
3. the static begin event corresponding to the `taskSwitch` action is retrieved in order to create a request for the quantity manager
4. first request requires that the begin event being executed now
5. second request requires that the end event (`event->get_other()` returns the end/begin events for begin/end events respectively) being executed now plus one tick

6. request to the simulation library is performed by calling the execute of the action

The actions are handled by mm_code class (Section 2.8). An action can correspond to one statement, a block of statements or even an entire function.
Any event a process can fire must be registered in the enum_static_events function. The simulator forces the user specify all the events a-priori in order to build an event set structure before starting the simulation. This phase is called elaboration and it is used by the simulation engine to create relationships among different events. For example let's imagine a writeint action in process dummy on a functional side is mapped onto cpuwriteint on a process called CPU. The mapping netlist called mymap implements the following function:

```
void mymap::intersect(mm_synch &synch) {
    synch.add("dummy", "writeint",          "CPU", "cpuwriteint");
    // ...
}
```

Again the intersect function is a pure virtual function used by the simulation elaboration phase to create events relationships. The mechanism is hidden underneath the mm_synch class (see ??).
If a named event is not registered in enum_static_event the elaboration phase raises an error.

2.2 The mm_medium

Communication is modeled explicitly using objects that are called media. A medium is defined with a set of variables and functions. Values of the variables constitute states of the medium. The variables can be referenced only by the functions defined in the medium, and communication among processes is established by calling these functions to change or evaluate the states of media.
This mechanism has been chosen for two reasons:

1. it allows communication semantics to be modeled solely by specifying media, independent from process specifications
2. it allows an efficient synchronization access on shared resources

The mechanism that handles the medium access is shown in figure 8.

Consider two processes P_1 and P_2. P_1 requests access to the medium in order to set a variable to value x. Another process, P_2, is waiting on a guard condition that involves the same variable.
The synchronization mechanism involves the following core-lib elements:

– mm_port_base (see section 2.1
– mm_port_mgr (see section 2.1

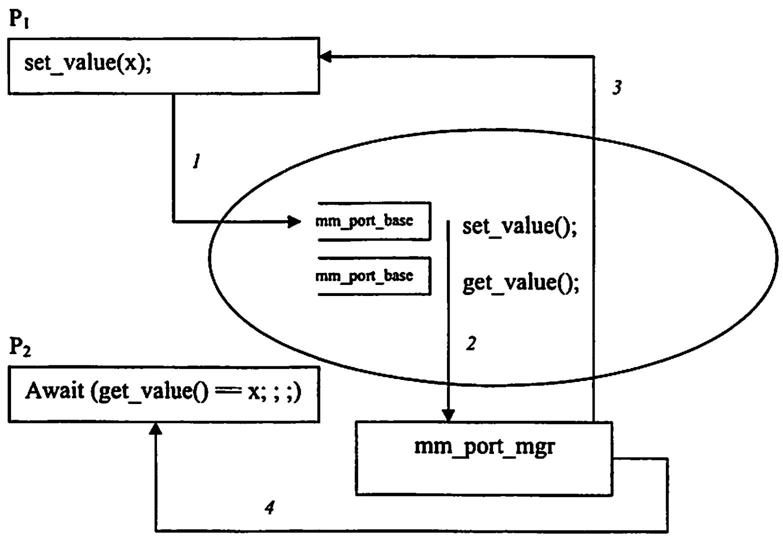


Fig. 8. Medium access protocol

The synchronization protocol is the following:

1. process P_1 requests access to the medium throughout a port derived from mm_port_base base class;
2. mm_port_base forwards the request to mm_port_mgr that atomically checks if the port is free and locks access to itself for different processes than P_1;
3. P_1 now writes the new value;
4. when P_1 releases the port lock the mm_port_mgr awakes all the processes (in the example P_2) that were waiting on that condition.

Internally core-lib relies upon Posix threads conditional variables to efficiently manage accesses. In particular the following Posix standard functions are used:

- `pthread_mutex_lock` and `pthread_mutex_unlock` to lock and unlock a mutex respectively;
- `pthread_cond_wait` and `pthread_cond_broadcast` to wait on and signal a conditional variable respectively.

If the port is not released deadlock may occur. The simulator provides some deadlock detection techniques that help the developer understand which process has locked a resource indefinitely (see

2.3 The mm_await

The `await` primitive is defined as follows:

```
mm_await await(this);
await.add(new mm_guard(myguard),
        new mm_portlist(mytestlist),
        new mm_portlist(mysetlist),
        new mm_code(mycriticalsection)
    );
    // ...
    await.execute();
```

where:

- myguard is any boolean expression (which may include function calls and in particular evaluate the states of media through ports);
- mytestlist is a set of ports
- mysetlist is a set of ports
- mycriticalsection is an action

The number n of quadruples myguard - mytestlist - mysetlist - mycriticalsection may vary ($n > 0$). The semantics is that the execution (`await.execute()`) of the construct blocks until any of the n `myguard` become true. Once at least one holds the ports listed in the corresponding `mytestlist` are tested to be unlocked and if this is true the ports listed in `mysetlist` are atomically locked. In this way while `mycriticalsection` is executing the states of the media listed in `mysetlist` may not be observable, i.e. no interface function can be executed for the media of `mysetlist` during the execution of `mycriticalsection`, except for those called directly by the thread executing `mycriticalsection`. Whether `mycriticalsection` is immediately executed or not when `myguard` is true depends on the scheduling constraints specified. The semantics guarantees that when the atomic section is started, and the `mysetlist` protects `myguard`, it remains so until the thread itself changes the states of medium.

The Semantics The semantics of the await is defined only if $guard_i$, $i = 1, \ldots, k$, is a state predicate and its evaluation involves no side effect. More specifically, a predicate over state variables can be defined with $guard_i$ and the following property holds: for any execution and for any state in the execution from which $guard_i$ is evaluated, the evaluation yields **true** if and only if the predicate holds at the state. Further, for all state variables of media and for all state variables of the process evaluating the $guard_i$ that are alive when the await statement starts, the evaluation does not change their values. In the sequel, we treat $guard_i$ as a propositional symbol that assumes **true** at a given state if and only if its predicate holds. We define **default** as a synonym of $(!guard_1 \&\& \cdots \&\& !guard_k)$, and thus exclude it from consideration in this section.

Given *PortName.TestList*, the meta-model uniquely defines a set of statements S(*PortName.TestList*). Similarly, a set of statements S(*PortName.SetList*) is defined for a given *PortName.SetList*. Exactly how these sets are defined will be described in Section 2.3, but the semantics is defined with these sets given, and how to compute them is not relevant. The semantics can be defined using the linear temporal logic given in Section ??. For this purpose, let us re-write the await statement with labels attached:

```
block(aw){
   await {
      (guard_1; PortName.TestList_1; PortName.SetList_1) block(at_1){statements_1; }
                         ⋮
      (guard_k; PortName.TestList_k; PortName.SetList_k) block(at_k){statements_k; }
   }
}
```

The label aw is attached to the compound statement that is the body of the await statement, while $\mathbf{at}_i$ is attached to the compound statement $\{$*statements*$_i$; $\}$ for each $i = 1, \ldots, k$. In the sequel, we may call $\{$*statements*$_i$; $\}$ a *critical section* of the await statement.

Let p be a process object that executes the await statement. The execution is subject to the following four constraints.

(a) $\mathbf{G}(\mathbf{pc}(p) = \mathbf{beg}(\mathtt{aw}) \Rightarrow ((\mathbf{pc}(p) \neq \mathbf{end}(\mathtt{aw}) \wedge \bigwedge_{i=1}^{k} \mathbf{pc}(p) \neq \mathbf{beg}(\mathbf{at}_i)) \;\; \mathbf{U} \;\; \bigvee_{i=1}^{k} \mathit{guard}_i \wedge \mathbf{pc}(p) = \mathbf{beg}(\mathbf{at}_i)))$

Intuitively, this constraint means two things. First, when the compound statement $\{$ *statements*$_i$; $\}$ starts, the expression *guard*$_i$ must be **true**. Second, the execution of the await statement cannot continue until some $\{$ *statements*$_i$; $\}$ starts, i.e. the await statement is blocked until then.

(b) $\mathbf{G}(\bigwedge_{i=1}^{k}(\mathbf{pc}(p) = \mathbf{end}(\mathbf{at}_i) \Rightarrow (\mathbf{pc}(p) = \mathbf{end}(\mathbf{at}_i) \;\; \mathbf{U} \;\; \mathbf{pc}(p) = \mathbf{end}(\mathtt{aw}))))$

Intuitively, this constraint says that if the compound statement $\{$ *statements*$_i$; $\}$ exits, the only way to continue the execution is to exit the await statement. Namely, it is not possible to execute two or more critical sections successively.

(c) $\mathbf{G}(\bigwedge_{i=1}^{k} \bigwedge_{\text{process } q : q \neq p} \bigwedge_{s \in S(\mathit{PortName.TestList}_i)} \mathbf{pc}(q) = \mathbf{beg}(s) \Rightarrow (\mathbf{pc}(p) \neq \mathbf{beg}(\mathbf{at}_i) \;\; \mathbf{U} \;\; \mathbf{pc}(q) = \mathbf{end}(s)))$

This constraint guarantees that when the critical section $\{$ *statements*$_i$; $\}$ starts, no other process object is executing any statement given in $S(\mathit{PortName.TestList}_i)$.

(d) $\mathbf{G}(\bigwedge_{i=1}^{k}(\mathbf{pc}(p) = \mathbf{beg}(\mathbf{at}_i) \Rightarrow (\bigwedge_{\text{process } q : q \neq p} \bigwedge_{s \in S(\mathit{PortName.SetList}_i)} \mathbf{pc}(q) \neq \mathbf{beg}(s) \;\; \mathbf{U} \;\; \mathbf{pc}(p) = \mathbf{end}(\mathbf{at}_i))))$

This constraint designates an exclusion of some actions by other processes while p executes a critical section. Namely, while $\{$ *statements*$_i$; $\}$ is executed, no other process object q executes any statement given in $S(\mathit{PortName.SetList}_i)$. This is different from the previous constraint in that it prevents other process objects from starting to execute certain statements, while the previous one implies that if other process objects have already started certain statements, then p cannot start to execute the critical section.

S(PortName.TestList)* and *S(PortName.SetList) In this section we define the sets S(*PortName.TestList*) and S(*PortName.SetList*). Intuitively, they represent the set of functions of interfaces specified in the list that are implemented in the objects designated by the ports, as well as critical sections of await statements whose *PortName.SetList* include common interfaces and objects designated by the ports.

Recall that *PortName.TestList* is a list of pairs made of a port *PortName* and an interface *IfName*, where *PortName* may be the keyword **this** and *IfName* may be the keyword **all**. If *PortName.TestList* were the keyword **all**, it designates the list made of *PortName.IfName* such that *PortName* is a port of the object in which the await statement resides and *IfName* is the keyword **all**.

Consider a pair *(PortName, IfName)* in the list designated by *PortName.TestList*. If *PortName* is the keyword **this**, then let o be the object in which the await statement resides. Otherwise, consider the object designated by the value assumed by the *PortName*. Without loss of generality, we assume that ports are connected and thus the object is defined. This object can be either a medium or netlist. The latter happens when the type of the *PortName* is the special interface Scope, and let o be the netlist in this case. Consider the case where the object is a medium. If a refinement of the medium has been specified, let o be the netlist representing the refinement. Recall that a refinement of the medium must be specified with a statement **refine**(*medium*, *netlist*), and we let o denote the second argument of this statement. If a refinement has not been specified for the medium, then o designates the medium itself.

Using the object o given above for the pair *(PortName, IfName)*, we define a set of pairs, each made of a medium object m and an interface f. Let us denote the set by M_f(*PortName,IfName*). Consider the case where o is a medium object. Then (m, f) is in the set M_f(*PortName, IfName*) if $m = o$ and $f =$ *IfName*. In case *IfName* is the keyword **all**, (m, f) is in the set if $m = o$ and f is implemented in m. In case o is a netlist object, (m, f) is in M_f(*PortName,IfName*) if m is a component of o, $f =$ *IfName*, and f is implemented in m. In case *IfName* is the keyword **all**, (m, f) is in the set if m is a component of o and f is implemented in m. We denote by M_f(*PortName.TestList*) the union of M_f(*PortName, IfName*) over all the pairs *(PortName, IfName)* in the *PortName.TestList*. Similarly, M_f(*PortName.SetList*) denotes the union of M_f(*PortName, IfName*) over all the pairs *(PortName, IfName)* in the *PortName.SetList*.

Now, using the sets M_f given above, S(*PortName.TestList*) and S(*PortName.SetList*) are defined as follows. They are defined in exactly the same way, i.e. if *PortName.TestList* and *PortName.SetList* are identical, so are the two sets S. We therefore consider S(*PortName.TestList*) only. A compound statement s is in S(*PortName.TestList*) if and only if either of the following two conditions holds. First, there exists (m, f) in M_f(*PortName.TestList*) such that s is the entire body of a function of the interface f in the medium object m. Second, s is a critical section of an await statement such that M_f(*PortName.SetList_1*) intersects with M_f(*PortName.TestList*), where *PortName.SetList_1* is the list specified at the third argument inside the parenthesis that precedes the critical section.

Implementation The mm_await class offer two main functions to accomplish the synchronization task. The prototype is the following:

```
void add(mm_guard *guard, mm_portlist *testlist, mm_portlist *setlist, mm_code *cs);
mt_base *execute(mt_base *param = NULL);
```

where the add parameters mean:

- mm_guard: is a class that holds either a pointer to evaluation callback function (to test the value of a variable dynamically) or a simple boolean function;
- mm_portlist: is an utility class that unfolds a list of ports of type mm_portbase into a STL (standard template library) list of ports for easy management;
- mm_code: is the critical section (see 2.8)

The execute function of the await is the same of the execute function of many other Metropolis building blocks. It accepts as input any types-lib basic type and produces as output any types-lib basic type. The execute algorithm is the following:

```
int totcount = _fire_mgr->save_status();
// deadlock detection
int iDeadLockCount = 0;

// loop until a critical section can run
do {
    int count = totcount;
    bool bAllHaveTestlist = true;
    bool bAtLeastOneGuardTrue = false;

    for (awaitlist_t::iterator it = _awaitlist.begin(); it != _awaitlist.end(); it++) {
        await_t *await = (await_t *)*it;

        if (bAllHaveTestlist) bAllHaveTestlist = !await->testlist->_portlist.empty();

        if (!guard_is_true(await, param)) {
            if (await->cs != NULL) await->cs->set_ps_dontrun();
            if (--count == 0) break;
            continue;
        } else
            bAtLeastOneGuardTrue = true;
            if (!mm_port_mgr::get_instance()->lock(_fire_mgr, await->setlist->_portlist)) {
                if (await->cs != NULL) await->cs->set_ps_dontrun();
                if (--count == 0) break;
            } // end if
        } // end for

        if (count > 0) {
            return _fire_mgr->execute_sched(param);
        } else if (bAllHaveTestlist && !bAtLeastOneGuardTrue) {
            pthread_mutex_lock(&_mutex);
                while (!_bSignal && result == 0)
                    result = pthread_cond_timedwait(&_cond, &_mutex, &abstime);
                _bSignal = false;
            pthread_mutex_unlock(&_mutex);
        } // end if

        _fire_mgr->restore_status();

    } while (true);
```

The first step requests the mm_fire_mgr attached to the current process to produce a snapshot of the events. The status is saved in the mm_fire_mgr itself in a stack structure to allow different nested awaits. The return value is the number of events that has a CANRUN status. Then main loop starts until a critical section is selected to run.
The critical sections are entered in a random order to optimize the lock/unlock mechanism of the testlists and setlists.
The semantics requires that if the guard is true and the ports in the testlist are free then the ports in the setlist are locked and the critical section is executed. Since these checks along with locks/unlocks must happen atomically they have been protected by a critical section in the mm_port_mgr. The random order is necessary to allow non-determinism since the first critical section that obeys the former constraints (the order is given by a list) is selected to run (the flag bSelected is set to true).
The flags bAllHaveTestlist and bAtLeastOneGuardTrue are used to decide whether or not enter the wait status in case no critical section is enabled. The idea is the following. If the user specifies a guard condition that evaluates a variable found in a medium (only a medium variable can change since it requires another active element - process to modify it) he is also required to protect the concurrent access to it with a suitable use of the ports. In particular in the testlist the port will be listed to

check if at the same time another process is modifying it. If the user provides the required `testlist` for each critical section then the flag `bAllHaveTestlist` is set to true. The flag `bAtLeastOneGuardTrue` is used to discriminate the fact that a critical section cannot run due to lock constraints in the `testlist` and not in the guard condition.
With a particular combination of these two conditions the process is safely put to sleep until another process wakes it again by modifying one of the conditions it was waiting for. The mechanism used is based on the Posix conditional variables. Each await registers its own conditional variable in the corresponding port. Whenever the port is released by another process to signal a status change in the medium, all the registered awaits are signalled by the `pthread_cond_broadcast` function.
The wait function used to put the process to sleep is the `pthread_cond_timedwait` that allows to specify a timer. This is useful to resolve deadlocks (see ??).
The status is restored whenever the await statement has no critical section enabled (blocks) to repeat the previous algorithm again.

2.4 The mm_port_mgr

The port manager `mm_port_mgr` is responsible for handling port requests in conjunction with the await statement by providing the necessary access control management.
Despite its name, the port manager is a passive element. It is invoked by the different processes that require access to the ports.
The initial configuration is a port manager per medium as in the figure below.

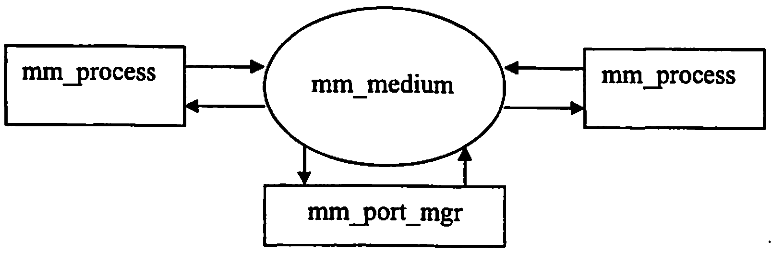


Fig. 9. Port manager

Consider two processes as in the figure above: P_1 and P_2. They may reach an await statement with different timings. When P_1 and P_2 reach the lock function call to obtain port access the simulator must ensure atomicity. It must avoid different processes take a wrong decision due to the distributed nature of the algorithm. This is achieved with a dynamically reconfigurable mutex. This mutex, at simulation startup, is created per port manager. As the simulation runs, whenever the situation below happens, where two processes P_1 and P_2 lock ports of different medium , the mutex is coalesced to guarantee atomicity.

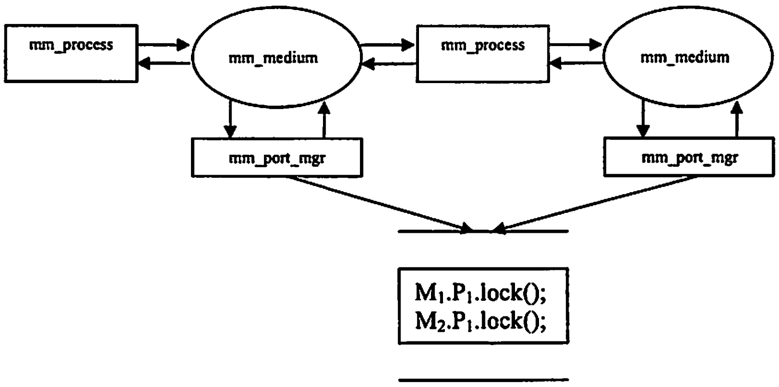


Fig. 10. Port manager with coalesced mutexes

2.5 The mm_fire_mgr

The fire manager contains the scheduling algorithm offering to Metropolis metamodel synchronization blocks two basic APIs as shown in figure:

- `execute_sched()`
- `execute_now()`

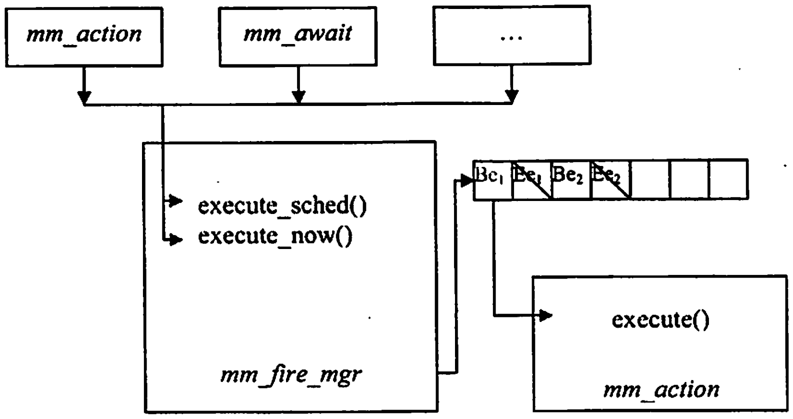


Fig. 11. Port manager with coalesced mutexes

Both `execute_sched` and `execute_now` run the scheduling algorithm before executing the action associated with the fired event. The only difference between the two function lies on the selection of the

events: execute_now accepts as input an event that is set to CANRUN state while all the other events are set to DONTRUN state; execute_sched is used when more than one event is enabled and the intersection algorithm (for synched events) is required.
The overall algorithm works as follows. When a synchronization function is reached (await, action, etc ...) the call to execute function is eventually redirected to either execute_sched or execute_now. Since the fire manager is executed in the current context of the invoking thread it has to synchronize with other threads before taking a decision. This task is carried out by mm_lxi_mgr and mm_gxi_mgr. The underneath synchronization algorithm is explained in section 2.6.
Suppose two processes have to be synchronized (i.e. they have synched events). When they reach the execute function (i.e. await.execute() or action.execute()) they call one of the execute_sched or execute_now function. The call is propagated in the simulator layers up to mm_gxi_mgr as in the figure below.

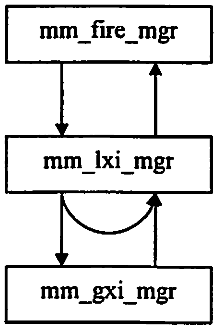


Fig. 12. Execute call stack

The mm_gxi_mgr registers the request and put the thread to sleep until the other thread reaches the same status. At this time all the events of both processes can be either in CANRUN or in DONTRUN state. The intersection algorithm is performed in mm_fire_mgr by calling the `resolve()` function. On the contrary, if a process has no synchronization constraints, it is allowed to take a decision on its own without forwarding the request up to the mm_gxi_mgr.
The intersection algorithm is based on an internal structure containing the pair list of static events (begin/end event) and the corresponding static action. Each static event has a runtime flag associated with it: the process status. It can take different values: UNKNOWN, CANRUN, DONTRUN and RUNNING.
Intersection is performed on the events that have CANRUN process status. Since different synchronization constraints are allowed the algorithm is based on an action automata. In the figure below (figure 13) three examples of action composition is shown:

- serial composition
- parallel composition

– mixed composition

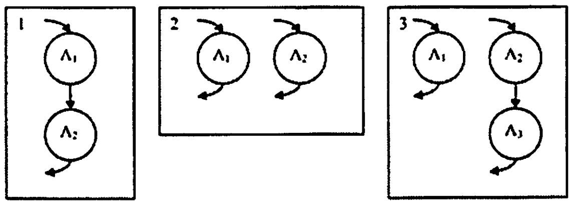


Fig. 13. Synchronization

Each node (vertex) in the graph of the action automata is called a state. Edges (the arrows) are associated with state transitions (events). Entering edges represent begin of an action while exiting edges represent end of an action. These are imagined to occur from instant to instant as the machine receives events in the input stream. Many action automata may be active for a single action if it is synched with different actions. The algorithm keeps accepting state transitions by setting un-acceptable events (edges) to `DONTRUN`.
The implementation of the action automata has been divided in two different classes:

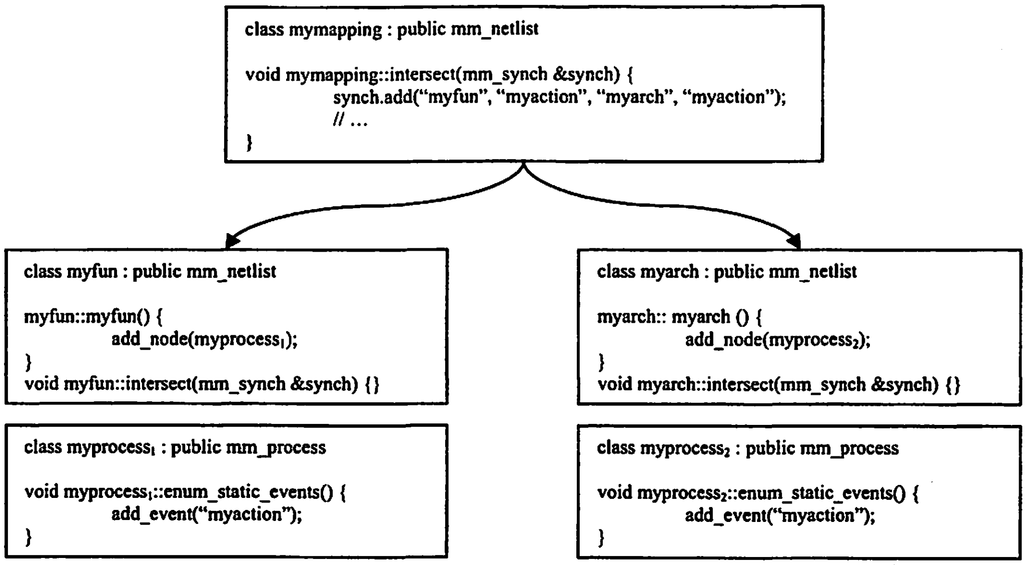
– the mm_event class where the action automata is stored;
– the mm_automata class where the algorithm is performed.

The action automata is built during the elaboration phase. The interface function call responsible for building the automata is the `intersect` member function of class `mm_netlist`. The default function body is empty. On the contrary, this function is always implemented in mapping netlists. Anyway, no assumption is made among different netlists (functional, architectural, mapping). It accepts a parameter of type mm_synch that offers a number of synchronization functions to allow different compositions among the actions.
In the figure below is shown an example of such synch constraint.

The mm_synch class may accept also a string as input where the string represents a LTL formula. The LTL formula is parsed by an internal parser and the action automata is built for it.
The fire manager execute function is written below:

```
do {
    xi = get_lxi_mgr()->request();
} while (_runevent == NULL);


// register the event
_event_mgr->add_gei(_runevent->get_fullname(), xi);
```



```
class mymapping : public mm_netlist

void mymapping::intersect(mm_synch &synch) {
        synch.add("myfun", "myaction", "myarch", "myaction");
        // ...
}
```

```
class myfun : public mm_netlist

myfun::myfun() {
        add_node(myprocess$_1$);
}
void myfun::intersect(mm_synch &synch) {}
```

```
class myarch : public mm_netlist

myarch:: myarch () {
        add_node(myprocess$_2$);
}
void myarch::intersect(mm_synch &synch) {}
```

```
class myprocess$_1$ : public mm_process

void myprocess$_1$::enum_static_events() {
        add_event("myaction");
}
```

```
class myprocess$_2$ : public mm_process

void myprocess$_2$::enum_static_events() {
        add_event("myaction");
}
```

Fig. 14. Synchronization

```
// call the debugger if active
if (mm_dbg_server::is_started() && _dbg_mgr) _dbg_mgr->execute(_runevent);

// if this event is an MM_BEGIN_EVENT it should have an attached action (mm_code)
if (_runevent->has_code()) {
    _runevent->set_ps(MM_PS_RUNNING);                  // set the event to MM_PS_RUNNING
    push_stack();                                      // <-- save current status
    result = _runevent->get_code()->execute(param);    // --- execute ---
    pop_stack();                                       // --> restore the status
} // end if
```

The while loop is responsible for executing the intersection algorithm. It exits only when one event (_runevent != NULL) has been selected. When this happens the simulator assign the tag to the event represented by the execution index (xi). Both the event and the execution index are stored in the event manager class (mm_event_mgr) in order to allow the user request events with a specific execution index. Next operation performed is verifying if a client debugger has been connected to the current process. If a debugger is connected, the process stops execution and control is passed to the attached debugger (for more info see ??).

The action is carried out in the end. Since the simulator accepts events without actions it first verifies if an action has been provided. If this is the case it first changes the status into RUNNING, then it saves the stack and eventually executes it.

2.6 The mm_lxi_mgr and the mm_gxi_mgr

The local execution index and the global execution index are responsible for allowing processes continue their execution independently and re-synchronizing them whenever a condition requires the local execution index be the same.

As anticipated in the previous section, this can happen when two processes have synched events or when a request to global time quantity manager is issued on a time annotation belonging to an event of another process.

The global execution index manager coalesces local execution index managers that require to be synchronized. This process happens during simulation as shown in the figure below (figure 15).

Consider two processes P_1 and P_2. The mapping netlist has the intersection function implemented

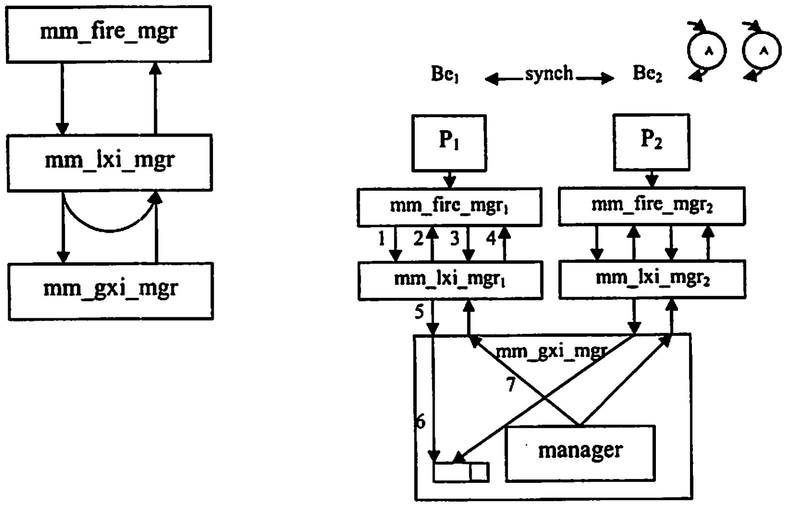


Fig. 15. Local/global execution index algorithm

requiring both of them being synchronized with respect to the events e_1 and e_2.

When they reach a synchronization point they request the fire manager that forwards the request to the local execution index manager. The local execution index manager (step 1) is executed in the current context of the calling thread since each process has its own local execution index manager. The manager requests the list of the events with the CANRUN flag back to the fire manager (step 2). Once the list is retrieved (step 3) the local execution index is able to check whether or not it is able to proceed on its own (thus executing step 4) or forward the request down to the global execution index

manager (step 5 and 6). Before forwarding (step 5) the local execution index calls the async2sync function to coalesce the two local execution index involved. Symmetrically process P_2 will execute the same steps, being completely unaware of the decisions taken by process P_1. Therefore the coalescing request will be recorded twice but executed only once.
Eventually the requests are forwarded to the global execution index manager whose main function is:

```
// the request from incoming mm_lxi_mgr requires:

// 1 - search the mm_lxi_mgr in the sets (extract the gxi where it is)
pthread_rwlock_rdlock(&_mutex);
    gxi_it_t set = find_lxiset(lxi_mgr);

    // if the process does not require gxi exit immediately
    if (set == (gxi_it_t)NULL) {
        pthread_rwlock_unlock(&_mutex);
        return false;
    } // end if

    gxi_t *gxi = (gxi_t *)set->second;
pthread_rwlock_unlock(&_mutex);

// 2 - forward the request to the thread
pthread_mutex_lock(&_mutex_req);

    do {
        // decrement the count so when it reaches 0 it means all mm_lxi_mgr are waiting resoluti
        gxi->req_count --;

        // forward the request to the thread if all mm_lxi_mgr are ready
        if (gxi->req_count == 0) {
            // update the pointer to pending request
            _gxi_req = gxi;
            // signal new request
            pthread_cond_signal(&_cond_new_req);
        } // end if

        // wait for the manager acknowledge to continue execution
```

```
            pthread_cond_wait(&gxi->_cond_ack, &_mutex_req);

            // the condition variable above can be signalled also when a merging operation
            // has been performed in the waiting set. In this case the pointer pMergedGxi
            // is not null
            if (gxi->pMergedGxi) {
                // record the destination structure pointer
                gxi_t *newgxi = gxi->pMergedGxi;
                // increment the count so that when all waiting mm_lxi_mgr have done
                // the last one free the memory
                if (++ gxi->req_count == gxi->lxiset.size()) delete gxi;
                // copy back and re-issue the request to the manager
                gxi = newgxi;
            } else
                break;
        } while (true);

    pthread_mutex_unlock(&_mutex_req);

    return true;
```

The local execution indexes are organized in sets. Each set contains the coalesced local execution indexes. When a process requests the global execution index to be synchronized the global execution index checks inside the sets for the gxi_t structure. If not found it means the process is free to continue and the function returns false. If found, the structure is the following:

```
    struct gxi_t {
        lxiset_t lxiset;                // the set of lxi_mgr
        unsigned int req_count;         // the number of pending requests the gxi expect before exec
        gxi_t *pMergedGxi;              // the pointer to the coaleshed gxi
        pthread_cond_t _cond_ack;       // conditional variable to awake waiting mm_lxi_mgr
    };
```

where

- lxiset contains the set of mm_lxi_mgr;
- req_count is the number of requests/mm_lxi_mgr/processes that still need to reach a synchronization point (thus a decision cannot be taken yet);
- pMergedGxi is a pointer to another gxi_t structure that will contain the coalesced structures;

– _cond_ack is a pthread conditional variable used to suspend the executing thread until all processes involved in the set reach the same point.

In order to manage all the requests a separate thread called manager is created along with mm_gxi_mgr class. The code executed is the following:

```
do {
    pthread_mutex_lock(&me->_mutex_req);
        while (!me->_gxi_req || (me->_gxi_req && me->_gxi_req->req_count > 0))
            pthread_cond_wait(&me->_cond_new_req, &me->_mutex_req);
        gxi_t *gxi = me->_gxi_req;

        // request is fullfilled -> reset it
        me->_gxi_req = NULL;
    pthread_mutex_unlock(&me->_mutex_req);

    // call the resolve of all the coalesced fire managers
    for (lxiset_t::iterator it = gxi->lxiset.begin(); it != gxi->lxiset.end(); it++) {
        mm_lxi_mgr *lxi_mgr = it->second;
        lxi_mgr->get_fire_mgr()->resolve();
    } // end for

    pthread_mutex_lock(&me->_mutex_req);
        // once done reset the counter
        gxi->req_count = (unsigned int)gxi->lxiset.size();

        // awake the waiting threads
        pthread_cond_broadcast(&gxi->_cond_ack);
    pthread_mutex_unlock(&me->_mutex_req);
} while (true);
```

This thread is kept running along all the simulation and does the following:

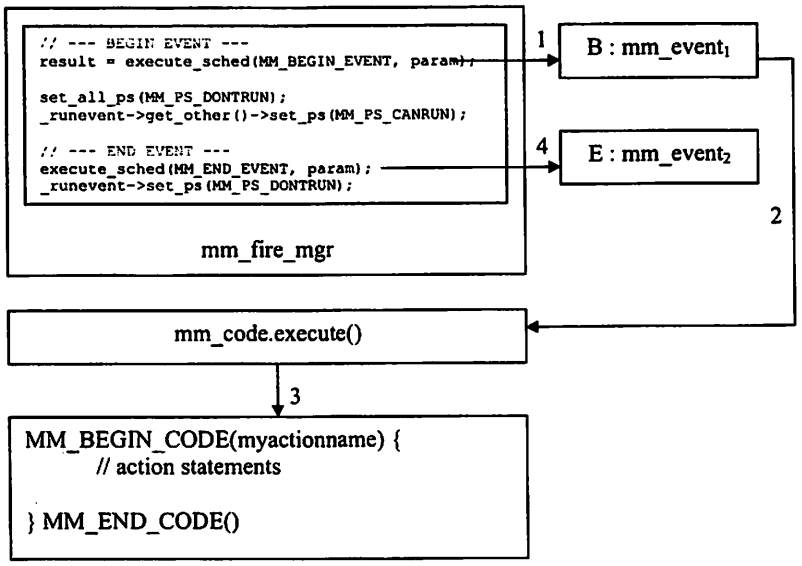
– it waits on a new request signalled with _cond_new_req conditional variable;
– it stores the request (the pointer to the gxi_t structure) and resets it;
– for each fire manager/local execution index it calls the resolve function to make the actual intersection of the events;
– awake all the waiting processes by broadcasting the _cond_ack conditional variable.

2.7 The mm_event

This class is responsible of handling a single event. It contains the information needed to perform a number of tasks (i.e. synchronization). An event belongs uniquely to a fire manager and therefore to a process. The same event can be nested in case of recursive functions or alike. The class has a stack to pile up event status. The pair begin/end defines an action.
To speed up execution and keep the structure clean the fire manager does not have the notion of action. It keeps information on events. The action is associated with the begin of the event and is represented by mm_code (??code). The end event does not have an action.

2.8 The mm_code

When the fire manager fires an event it calls the action stored in the mm_event class. The action is stored as an instance of class mm_code. Not all the events have an action stored. In the figure below is shown a typical execution.



```
// --- BEGIN EVENT ---
result = execute_sched(MM_BEGIN_EVENT, param);

set_all_ps(MM_PS_DONTRUN);
_runevent->get_other()->set_ps(MM_PS_CANRUN);

// --- END EVENT ---
execute_sched(MM_END_EVENT, param);
_runevent->set_ps(MM_PS_DONTRUN);
```

```
MM_BEGIN_CODE(myactionname) {
        // action statements

} MM_END_CODE()
```

Fig. 16. Class mm_code

The fire manager fires the begin of an event (step 1). Since the mm_event class has a pointer to the mm_code associated, the fire manager first verifies if the pointer is not null and then calls the `execute()` function (step 2) of class mm_code. Inside class mm_code is stored a pointer to the callback function that implements the action.
The callback mechanism is a general mechanism used in many cases when one layer of software needs

to get services from higher layers without hurting. The layer which needs the service only knows (and actually declares) the prototype of the service required, not the actual service provider itself. The address of the function which will actually perform the required service is supplied at run-time, via some registration mechanism. In metropolis framework the supplied run-time function is registered with the following snippet of code:

```
add_static_event("myevent", &mycallback\_action);
```

where `myevent` is the event name and `mycallback_action` is the pointer to the action. This has been used extensively in C. The problem presents itself when C++ class methods are the functions you would like to perform the job, not "plain" functions as is the usual case. Two things need to be resolved: the first is how to get the address of the method into a `void*`. The second thing is how to call the method (the `this` issue) once the function address is known.

The first problem is solved with a trick: no cast will work, because all of them do some sort of type-checking. The compiler will not do any type checking on the actual parameters passed for an ellipses argument. This way, a really small function can be made to cast practically anything into a `void*`.

The second problem, of how to call the method once its address is known, is solved by using a "caller thunk": a function whose job it is to call the method pointed to by a `void*`, given the pointer and a value for this. Most methods' calling convention is `__stdcall`, which means amongst other things, that this is passed in the `ECX` register and that parameters are pushed on the stack from right to left. Using this knowledge, a small function in assembly invokes the `__stdcall` methods itself.

2.9 mm_netlist

The netlist is a simple container for:

- processes (see ??);
- mediums (see 2.2).

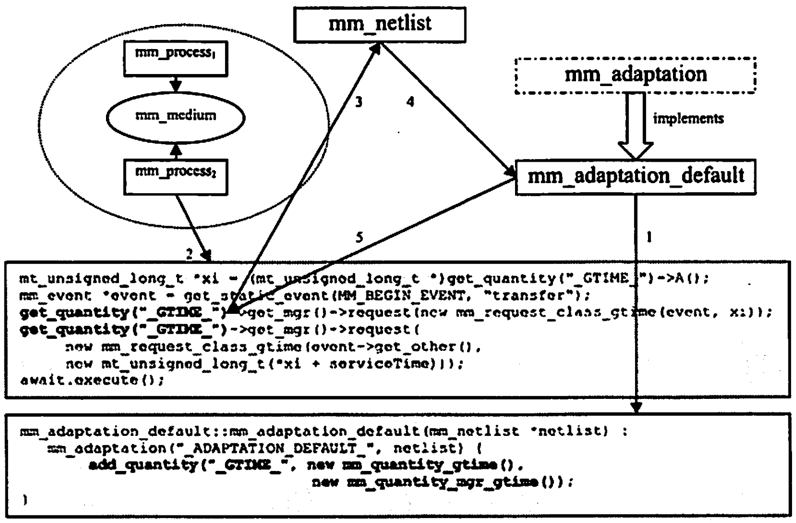
The API offered are the following:

- `add_node()` to add a node (process or medium);
- `find_node()` to find a node by its name.

In the constructor of the class `mm_netlist` the default behavior is instantiating the `mm_adaptation_netlist` to handle quantities and quantity managers.
The call to the `execute()` function call spawns all the processes declared in the netlist by starting executing the thread entry point.

2.10 The mm_adaptation and mm_adaptation_default

The adaptation netlist is a helper class that contains quantities and their managers. The default adaptation netlist is instantiated automatically whenever an `mm_netlist` object is created (step 1 in figure 17 shows the code of the constructor of the class `mm_adaptation_default`).



```
mt_unsigned_long_t *xi = (mt_unsigned_long_t *)get_quantity("_GTIME_")->A();
mm_event *event = get_static_event(MM_BEGIN_EVENT, "transfer");
get_quantity("_GTIME_")->get_mgr()->request(new mm_request_class_gtime(event, xi));
get_quantity("_GTIME_")->get_mgr()->request(
     new mm_request_class_gtime(event->get_other(),
     new mt_unsigned_long_t(*xi + serviceTime)));
await.execute();
```

```
mm_adaptation_default::mm_adaptation_default(mm_netlist *netlist) :
   mm_adaptation("_ADAPTATION_DEFAULT_", netlist) {
        add_quantity("_GTIME_", new mm_quantity_gtime(),
                                new mm_quantity_mgr_gtime());
}
```

Fig. 17. Class mm_adaptation

Once the default quantities are loaded each process or medium may refer to them by calling `get_quantity(name)` function call (step 3). The `get_quantity(name)` function call performs the following steps:

- it redirects the request to the containing `mm_netlist` class object (step 3);
- the `mm_netlist` has associated either the default adaptation netlist instance (at the moment it instantiates a global quantity by default) or a user-defined adaptation netlist instance; the library calls the `get_quantity(name)` function call of the registered adaptation netlist instance (step 4);
- the implementation of the `get_quantity(name)` function call inside the `mm_adaptation` class returns the `mm_quantity` object stored.

Once the `mm_quantity` is retrieved the process may access the manager by calling `get_mgr()` and queue the request by calling `request` with a parameter of type `mm_request_class`.
The `mm_adaptation` is an abstract class that exposes the following basic API:

- `add_quantity` adds a new quantity to the netlist;
- `get_quantity` returns the quantity by its name;

- `get_eventset` is a query interface towards all the fire managers belonging to the netlist and returns all the events that obey a particular search criteria.

The `mm_adaptation` requires the implementing adaptation netlist implement the following functions:

- `qm_resolve_needed()` is used to speed up simulation; it returns true to request the fire manager call the `qm_resolve()` function;
- `qm_resolve()` calls the resolving strategy of the managers registered with each quantity;
- `qm_is_stable()` returns true when a fixed point has been reached;
- `qm_postcond` is called after the resolution phase has ended.

3 Types-lib: the library of basic types

The `types-lib` library provides a set of fundamental types that are complementary to C++ basic types.
The library has been developed to serve as a high level communication mechanism between different adapters developed with the co-simulation framework presented in section ??.
In the co-simulation framework the use of interfaces has been highly standardized in order to allow two or more different domains interoperate among them. The interfaces specify:

- how each context identify itself;
- how data is transmitted between each context;
- remote procedures that the client application can call;
- data types for the parameters and return values of the remote procedures.

In this section focus is placed on data types. Data types undergo to the process called **`marshalling`** in order to be properly exchanged among the different domains. Marshaling is the process of packaging and un-packaging parameters so a remote procedure call can take place. In this section the term remote procedure call (RPC) is used because the mechanism involved resembles the one specified in RPC.
Different parameter types are marshaled in different ways. The `types-lib` hides the way each type is treated behind a standard interface that provides two basic functions:

- `toXml` that converts a basic data type into the corresponding xml representation (packaging);
- `fromXml` that converts back an xml representation into the corresponding type (un-packaging).

The `types-lib` class hierarchy is shown in figure 18.
All the basic types derive from:

- `mt_object` that provides the basic object naming mechanism (each variable has an associated string based name);

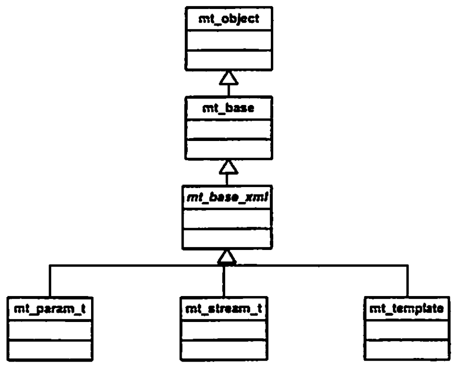


Fig. 18. Basic types library class hierarchy

- `mt_base` that allows type recognition facility (a C++ implementation of the java-like `instanceof` keyword);
- `mt_base_xml` that provides the above `toXml` and `fromXml` functions.

The types-lib fundamental types are:

- `mt_template` a generic template-based class from which integer, float, double, long, unsigned numbers are derived;
- `mt_stream` a base class for handling binary streams both for input and output;
- `mt_param` a generic structure able to accommodate sequences of the basic types (recursive structures are allowed).

4 Distributed Simulation Flow

In the figure below the simulation strategy is outlined for an number `n` of processes.

The overall algorithm does not use any central manager. Therefore the simulation flow is distributed. In the example `n` of processes are given. Each process has its own:

- `fire_mgr`, the component responsible for selecting and firing events;
- `lxi_mgr`, the component responsible for synchronizing distributed processes that require event intersection (i.e. in mapping) and related quantity resolution.

Each process (`mm_process`) starts execution in the `thread` function inherited by the `mm_process` base class. A process is composed by zero or more `atoms`. If no `atoms` are present it means that no events are exposed to the other processes (i.e. a process that runs on its own that does not require any synchronization or any quantity annotation). In this case the `enum_static_events()` function will be empty. Otherwise one or more `atoms` will be present and for each one the corresponding pair begin/end events will be listed in `enum_static_events()`. Two examples where the events of the `atoms` have to be exposed are:

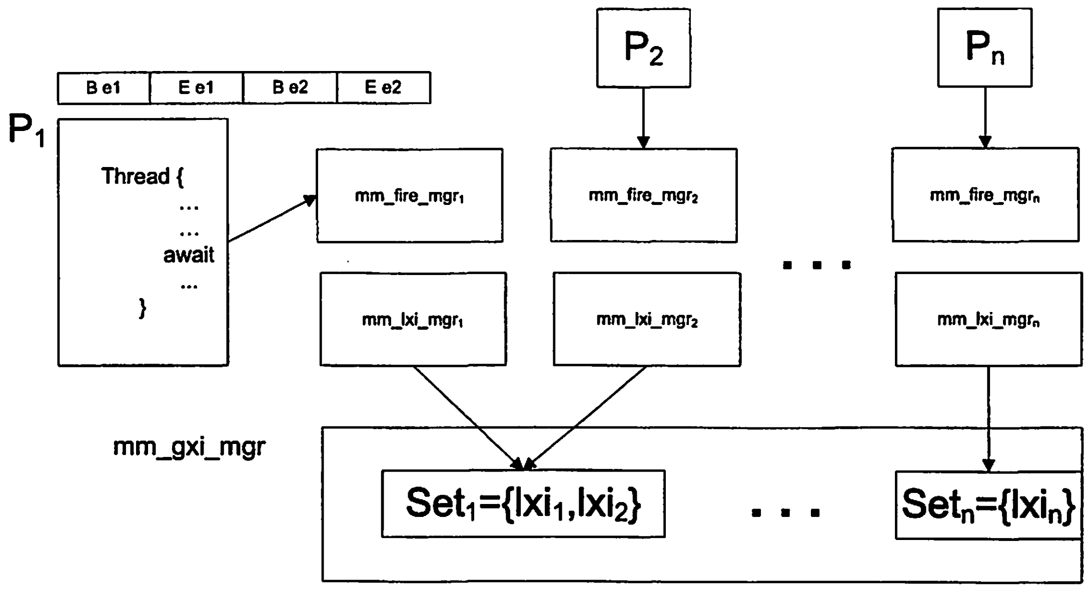


Fig. 19. Simulation flow

- the actions corresponding to the critical sections of the `await` statement;
- the pieces of code (a statement, a function or an entire algorithm) annotated with a quantity.

Suppose the following situation happens:

- P_1 hits an `await` statement;
- P_2 hits an `await` statement;
- P_3 executes a read from a fifo that requires 10 clock cycles.

Beneath suppose the events of some `atoms` of P_1 are synchronized with the events of some `atoms` of P_2.
Since P_1, P_2 and P_3 are running in parallel (no assumption is made on the hosting OS execution semantics) at a certain point they will hit the `execute()` function of the corresponding library component:

- `await.execute()` for P_1 and P_2;
- `action.execute()` for P_3.

The `execute()` function is called in the execution context of the calling process. It first calls the fire manager (each process has its own fire manager). The fire manager knows all the `atoms` that a process can execute since they are loaded at the beginning of the simulation (elaboration phase). According to the library component used (await, action, etc) the status of the events corresponding to the `atoms` are set to `CANRUN` or `DONTRUN` state. For example, suppose the `await` statement in P_1 is

```
await.add(new mm_guard(has_fifo_elements), NULL, NULL,
            new mm_code(read_element));
await.add(new mm_guard(has_fifo_space), NULL, NULL,
            new mm_code(write_element));
```

Suppose await statement in P_2 is

```
await.add(new mm_guard(true), NULL, NULL,
            new mm_code(cpu_read));
await.add(new mm_guard(true), NULL, NULL,
            new mm_code(cpu_decode));
await.add(new mm_guard(true), NULL, NULL,
            new mm_code(cpu_execute));
await.add(new mm_guard(true), NULL, NULL,
            new mm_code(cpu_write));
```

The static events in the fire managers of process P_1 and process P_2 are respectively:

- begin/end of has_fifo_elements, read_element and write_element;
- begin/end of cpu_read, cpu_decode, cpu_execute and cpu_write.

The guards may raise events as in this example if the corresponding atom is required to be mapped in a hardware component. It is up to the designer expose the events he is interested.
Before intersecting events, the simulator must ensure that all the processes involved in the intersection algorithm have reached a state where no atom is in the UNKNOWN state. This means that all the processes are about to execute the corresponding atoms (the begin of the atom). This mechanism is realized by the local execution index.
The local execution index:

- checks the events that can fire;
- builds the graph of related events (the relationships are dynamically loaded during elaboration phase by the mm_synch class);
- for each event pair it retrieves the fire manager with which it requires coordination and coalesces the corresponding sets by calling async2synch in the global execution index.

Otherwise, in case no set of the events require synchronization with any other event belonging to another fire manager (in the example process P_3), then the local execution index allows the fire manager continue the execution without waiting acknowledge from the global execution index manager. When the sets in the global execution index are coalesced, the process that has reached the intersection algorithm is halted waiting other processes reaching the same state (in the example P_1 and P_2 are coalesced and have to wait each other). When all the processes have selected the possible events to fire, the control is passed to the global execution index that calls the intersection algorithm of each fire manager (the fire manager of P_1 and P_2).

The function executed in the fire manager is called `resolve()` and does the following:

– it intersects the events. The event pairs are considered not fire-able if any of the two events is in `DONTRUN` state;

– it calls the quantity managers of the netlist to which the process belongs to. The quantity managers act on the events by setting to `DONTRUN` those that do not satisfy quantity constraints.

Since quantity managers are developed by the users, the framework provides a set of API to access event sets easily. To this aim a function called `get_events` accepts as parameter a filter to obtain different sets of events.

At the end of these steps only a few events can fire. The following may happen:

1. no events are in `CANRUN` state. The process is halted and the `execute()` function call does not return. The intersection algorithm along with quantity resolution phase are re-executed on the same set of fire-able events;
2. exactly one event is in `CANRUN` state. The `execute` function of the corresponding `action` is called and the statements in the `atom` get executed;
3. more than one event is in `CANRUN` state. The simulator picks up one randomly and calls the `execute` function of the corresponding `action`.

Since an `action` may be split in one or more `atoms`, the events may pile up recursively in a stack-like mechanism. Each firing keep the running `action/event` in a stack, in order to synchronize the end of the events properly.

Simulation ends when all the processes exit.

5 Event Debugger

The `MetroC` simulator provides an advanced debugger that differs from standard debuggers in many aspects. The main ones are summarized below:

– it allows to debug the entire system remotely (the core-lib provides a server mechanism to allow debug clients connect to itself);

– the number of clients depends on the number of processes in the system. The debugger allows each client connect to the process being debugged directly and independently from other clients (i.e. three different client debuggers may connect to a cpu, a memory and a bus to check their correct behavior);

– it allows to put breakpoints on single events, watch event stacks and event graphs (the relationships among events that are built during elaboration phase);

- it allows to refine netlists dynamically by connecting/disconnecting ports. In particular it allows to change the level of abstraction by selecting the system stack dynamically;
- it allows to act on non-deterministic choices by selecting corner-cases execution paths for critical applications;
- it allows to load/unload quantities and corresponding managers (i.e. it is possible to test different schedulers);
- it allows to communicate with the available quantity managers in order to optimize the managers and avoid fixed-point algorithm deadlocks (i.e. it is possible to modify the status of the events between CANRUN and DONTRUN and beneath that it is possible to force the `is_stable()` function result thus jumping out from the fixed point algorithm);
- it allows to take a snapshot of the system thus helping improving deadlock detection.

The way the debugger works is depicted in the figure below (figure 20).

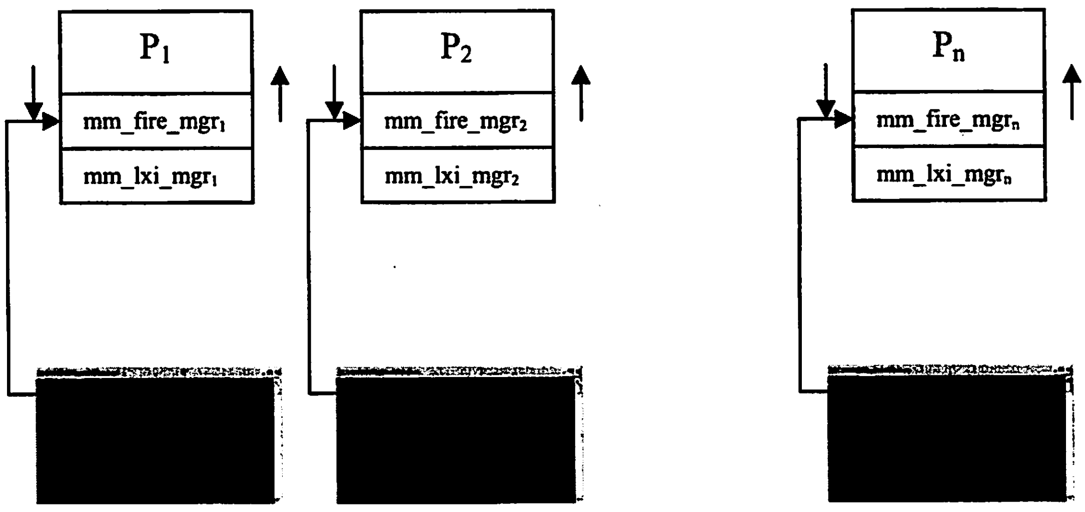


Fig. 20. Debugger overview

In the current implementation the debugger accepts a set of commands from a command-based shell.

The shell requires the user specify three connection parameters:

- the network address where the system is running;
- the network port to connect to (i.e. different network ports may be used in order to simulate a network of nodes on the same machine);
- the process identifier to connect to (i.e. cpu1 and cpu2 in a symmetrical multiprocessing system).

Once the connection is established a hooking engine is activated in the fire manager. Refer to figure 21.

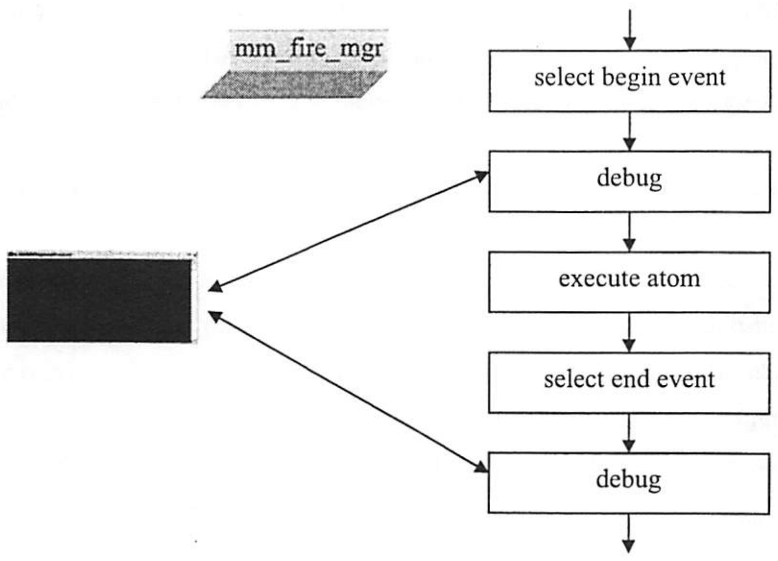


Fig. 21. Hooking engine in the fire manager

When the fire manager is invoked by a synchronization mechanism (await, label annotation, etc) it first selects the event to fire (see section 2.5 and section 4).
When the event is selected it then checks if a client debugger is attached. If it is attached (the process is being debugged), the debugger server code is executed in order to respond to the requests of the client. Control is passed to the debugger server running in the context of the process being debugged. Thus the control of the process is completely handled by the client debugger.
All the functionalities listed in the introduction of this section are achieved throughout debugging services offered by different subsystems that constitute the overall simulator.

5.1 Control simulation flow

The handling of events is processed in the fire manager. If the user puts a break point on an event, the system adds a flag in the event structure present in the fire manager that eventually breaks, passing control back to the client debugger along with all the necessary information (i.e. the status of all the events/atoms).
Other actions are possible such as break whenever no event is fire-able (possible deadlock) or break on a named event.
Events status can also be changed. If a non deterministic choice is possible (more than one event is fire-able) it is possible to select the event to fire. An example of this functionality is shown in the snippet of code below:

```
CPU 1> print events CANRUN
```

```
cpu_execute
cpu_read
cpu_write

CPU 1> set event ALL DONTRUN
CPU 1> set event cpu_read CANRUN
```

5.2 Change refinement level dynamically

It is possible to dynamically change level of abstraction by selecting different ports that correspond to the different levels.
When level of abstraction is changed all the processes are halted in order to avoid raising events that are somewhat correlated with the old platform stack. The mechanism is shown in the figure below (figure 22). When process P_1 calls a service of medium `my_level1_medium` it first calls an interface

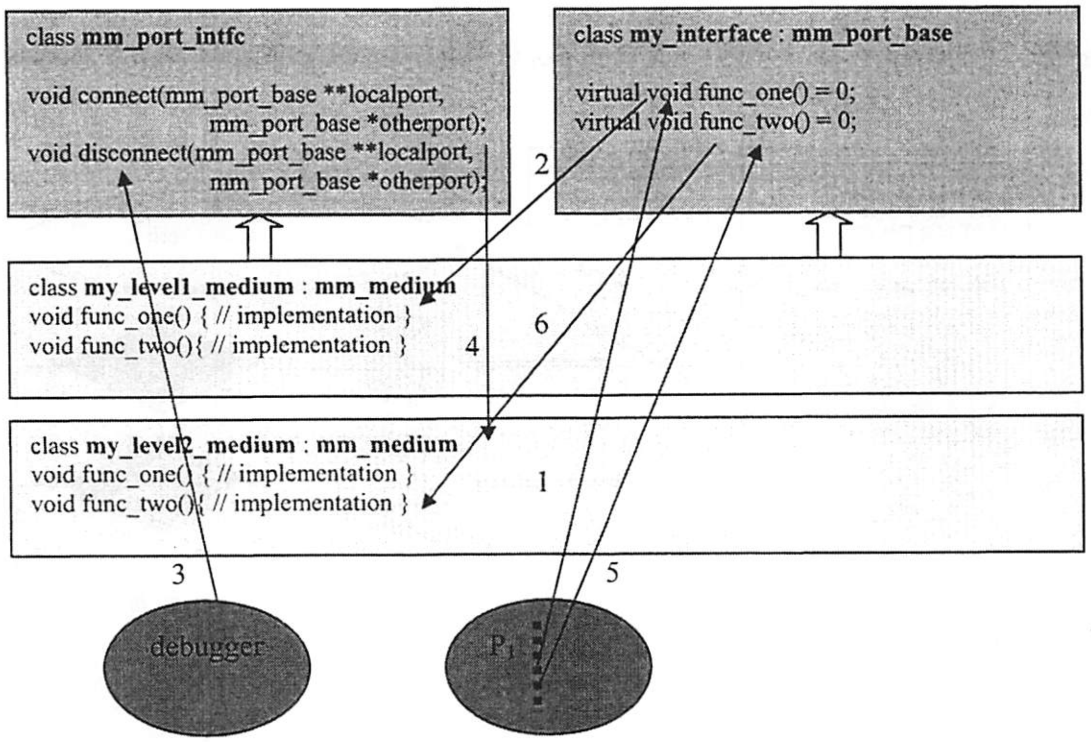

class mm_port_intfc
void connect(mm_port_base **localport, mm_port_base *otherport);
void disconnect(mm_port_base **localport, mm_port_base *otherport);
class my_interface : mm_port_base
virtual void func_one() = 0;
virtual void func_two() = 0;
class my_level1_medium : mm_medium
void func_one() { // implementation }
void func_two(){ // implementation }
class my_level2_medium : mm_medium
void func_one() { // implementation }
void func_two(){ // implementation }
1
2
3
4
5
6
debugger
P1

Fig. 22. Refinement procedure

that eventually redirects the call to the real implementation of the function (in the example arrow 1 shows the call to the interface and arrow 2 shows the call to the implementation of the function).

When the operations in the medium are executed in (the atom has finished), control is passed back to the fire manager that eventually returns to the main `thread` function of process P_1.

If a debugger is attached to P_1 and the user decides to change refinement level, as soon as the fire manager is called the process is halted in order to substitute `my_level1_medium` with `my_level2_medium`. This operation (arrow 3) first requires the debugger call `disconnect` (arrow 4) on all the ports dealing with P_1 and `my_level1_medium`. Then in the same operation the debugger issues a connect between the ports of P_1 and `my_level2_medium`. Once a connection has been re-established, simulation is allowed to continue. In the example, next during simulation, P_1 requests a service to the medium interface offered throughout the function `func_two()`. The entire system is now arranged in such a way that eventually this call gets redirected in the implementation of `my_level2_medium` (arrows 5 and 6).

5.3 Debug quantities and managers

Quantities and their managers are stored in `mm_adaptation_netlist` class that is automatically instantiated whenever a `netlist` is created.

In the figure below (figure 23) a typical `mm_adaptation_netlist` is shown along with the quantities that it handles. Standard quantity managers provide four common interface functions: `request` to

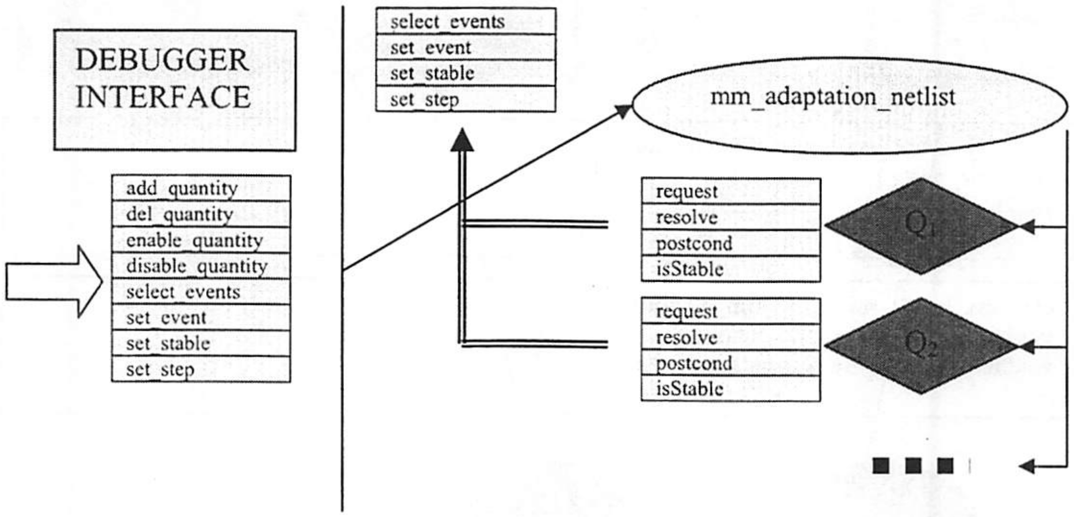


Fig. 23. Quantity managers debugging

request a quantity manager, `resolve` called by the framework to resolve the quantity, `postcond` called by the framework to do do some clean up jobs on the quantity, `isStable` called by the framework to end the fixed point convergence algorithm.

In order to provide debugging capabilities to the quantities and their managers a debugging class called mm_quantity_dbg_intfc has been added with the implementation of the necessary functions. By extending the mm_quantity with the previous class the following functions are inherited:

- select_events executes a query on the events related with the quantity. It allows to extract events that correspond to one of the following criteria: the events that can be fired (CANRUN state) and the events that cannot be fired (DONTRUN state);
- set_event set the status of an event (possible values are CANRUN or DONTRUN state);
- set_stable allows to change the return value of the isStable function thus interrupting the fixed point algorithm;
- set_step allows the user enable/disable steps execution of the quantity resolution algorithm. It is possible, for example, test the resolve function without applying changes with the postcond function.

An example of this functionality is shown in the snippet of code below:

```
CPU 1> quantity _GTIME_ select events CANRUN

cpu_execute

CPU 1> quantity _GTIME_ set step postcond disabled

OK, the postcond function of quantity _GTIME_ has been disabled
```

6 Deadlocks detection techniques

6.1 Await

The await statement is one of the most complex components due to the fact it is responsible for synchronization constraints. Since the processes run in parallel the await statement has been built in such a way no centralized manager is necessary.
The overall idea follows client-server mechanisms.
Consider figure (24). Whenever a process reaches an await statements it tests the guards and for those satisfied test lists are checked. The check of the test lists and the lock of the set lists is provided by the medium that acts like a server. Once the ports have been locked access to the process is provided throughout the port. The lock is kept until the critical section terminates.
During the different phases of the algorithm many different things can happen.
In the worst case no condition is satisfied and the await must block execution.
This status is reached in two different ways: by polling or by waiting on a conditional variable.
There are some conditions that allow the algorithm to put the process to sleep status and wait for a

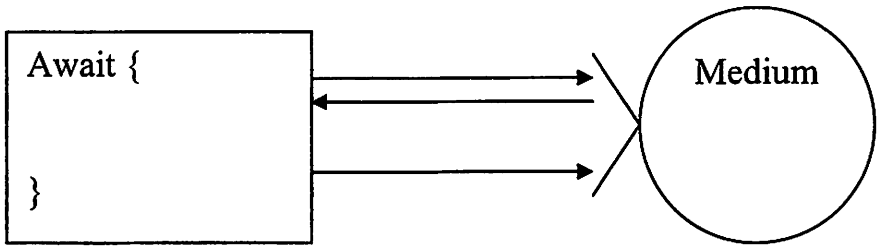


Fig. 24. System overview

condition to change.

These conditions rely upon user way of specifying await guards, test lists and set lists. In any case the most conservative choice is made to avoid deadlock.

The conditions are the following and can be specified with axioms.

1 If all the guards are false and at least one test list has been specified then the process can use conditional variable

2 If one or more guards are true then polling is necessary

3 If all guards are false but no test list has been specified then polling is necessary

Polling is necessary whenever the user does not provide a way to the simulator for understanding if any variable in the medium have been changed by another process. Typically this change is done after protecting the variable with a critical section throughout the use of ports.

Inside each port the simulator store a list of conditional variables (posix) that are waiting for a condition to be satisfied. As soon as a port is released it means that some of the states variables are changed and therefore interesting processes must be awaken to test if something has changed.

When multiple critical sections are present and many processes are queued in the ports then deadlocks may arise.

To help the user detect deadlocks a ranking deadlock mechanism have been implemented.

The ranking goes from 1 to 8 where lower values mean soft deadlock/time consuming while higher values mean hard deadlock.

A soft deadlock is a momentary deadlock due to the fact that test lists and set lists have not been properly set. If this happen the conditional variables interested by the change of a condition are not signalled and the corresponding thread is kept sleeping. The anti-deadlock mechanism awakes the thread after a certain amount of time and, if the conditions that prevented any of the critical sections to be executed have changed, it produces a warning for the user.

If all the conditions and test lists/set lists remain unchanged for a long period of time the thread is considered hard deadlocked and the warning is issued again.

When the warning is produced the system is able to create a snapshots of the ports usage in order to allow user understand what is wrong.

6.2 Fire manager/ Synch

7 Co-simulation framework

7.1 A software design pattern approach to hw/sw co-design

The key features of the hardware co-simulation framework are:

- it is a general framework since its main task is define communication interfaces that can be used at any level of abstraction
- it allows integration of different IP models
- it allows the user to do the mapping according to the ways he is used to
- it keeps orthogonalization between behavior and architecture
- it keeps the key-elements (i.e. quantities) abstracted from the actual implementation

The framework has been developed in a library.
In order to provide a general framework on one hand and ad-hoc solution for many of the available IP models on the other the library has been divided into two separate parts:

- a part that provides the basic communication primitives and the main interfaces
- a part that implements those interfaces for a particular IP block.

The first part is called hardware abstraction layer or adapter: it implements the media between core-lib and the different IPs. It provides the basic API and communication primitives without committing to a particular hardware implementation. The second part is the actual implementation that bridges a piece of code in Metropolis to the corresponding hardware. It allows different level of abstractions according to the selected medium and IP model.
Two design patterns have been used: the adapter and the bridge.
The adapter converts the interface of a class into another interface client expect. Adapter let classes (components) work together that couldn't otherwise because of incompatible interfaces. The participants are:

- the Target: it defines the domain-specific interface that Client uses
- the Client: it collaborates with objects conforming to the Target interface
- the Adaptee: it defines an existing interface that needs adapting
- the Adapter: it adapts the interface of Adaptee to the Target interface

Bridge has a structure similar to object adapter, but Bridge has a different intent: it is meant to separate an interface from its implementation so that they can be varied easily and independently.

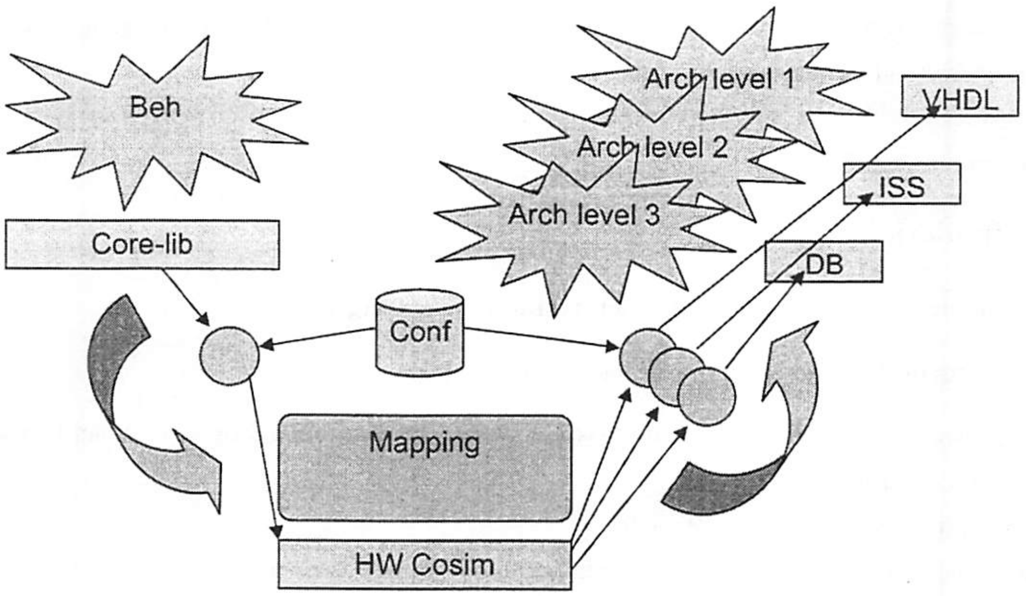


Fig. 25. Hardware co-simulation framework overview

7.2 Implementation

The implementation reflects the design methodology and mainly focus on define

- well established communication interfaces at both endpoints
- a framework for easy implementations of ad-hoc mediums/adapters for the target IP blocks

The communication paradigm used is similar to XML-RPC and it is shown in the figure below.

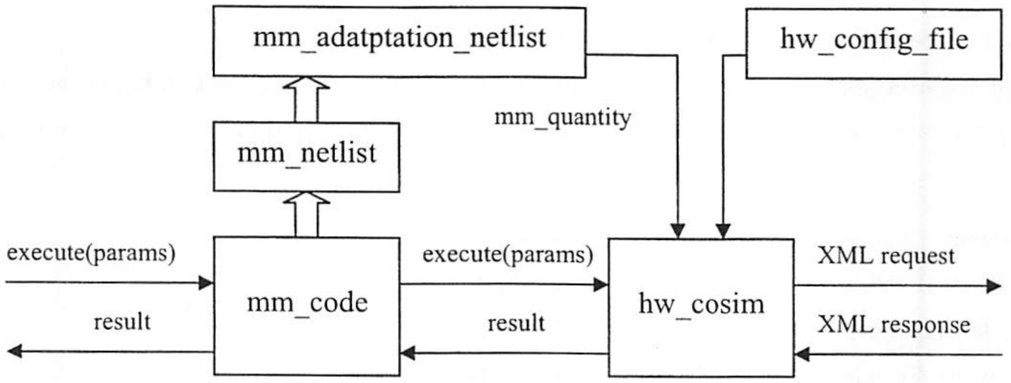


Fig. 26. Hardware co-simulation XML-RPC style communication

XML-RPC is a Remote Procedure Calling protocol that works over the Internet. An XML-RPC message is an HTTP-POST request. The body of the request is in XML. A procedure executes on the server and the value it returns is also formatted in XML. Procedure parameters can be scalars, numbers, strings, dates, etc.; and can also be complex record and list structures. In the co-simulation framework the XML-RPC request/response paradigm is maintained as shown in the figure. The local `execute()` function call is processed by the `hw_cosim` class that produces an XML stream with the remote function call and a number of parameters.

The way the remote procedure is invoked along with the way the parameters are collected differ from the XML-RPC standard.

In the co-simulation framework many RPC-style `execute()` functions may be invoked at the same time. For example in order to co-simulate a CPU, a memory and a bus the designer may want to use an ISS (instruction set simulator) for the CPU model and a VHDL model for the memory banks and the bus. The concurrent executions of these elements correspond to a number of parallel calls to the corresponding `execute()` functions.

Beneath this multiple communication issue there are some major differences in the way the parameters are collected and the type of information exchanged. These differences are shown in the figure above and involve the following Metropolis frameworks components:

- the netlist where the action is executed (`mm_netlist`);
- the adaptation netlist where the quantities belonging to the netlist are stored (`mm_adaptation_netlist`);
- the hardware configuration file (`hw_config_file`) where all the necessary information to adapt the local domain state to the remote domain state are stored (i.e. registry and memory mappings, etc ...).

When the `mm_code.execute(params)` is called the `hw_cosim` library does the following:

- it calls the `to_xml()` function of the `mt_param_t` class in order to marshal the parameters in XML;
- it loads the hardware configuration file and maps each parameter declared in `params` into the corresponding IP block element (registry, memory address, etc ...);
- according to the adapter end-point involved it may add some run-time information (i.e. working directory, base address, RTOS binary image, etc ...);
- it traverses the design hierarchy up to the netlist that contains the adaptation netlist where the quantities related to the action being executed are stored;
- for each quantity it calls the `to_xml()` on the annotation and stores its value in the XML packet that is going to be transmitted.

The first aim is reached inside the C++ class hw_cosim. This is the main class that handles co-simulation between the metamodel C++ mm_code class in core-lib and C++ hs_adapter class in the target implementation. The hs_adapter is an abstract class whose implementation is demanded to the ad-hoc medium-adapter and will be described later.

The hw_cosim class is responsible for the communication between the two endpoints and does the following.

The master the communication is the part that requests a service. In a common scenario this role is represented by the mm_code C++ class that embodies metamodel actions.

To initialize remote co-simulation the hw_cosim class exposes the very same interface of a standard metamodel action plus the possibility of specifying a configuration file that is used to handle target hardware (see).

When mm_code requests a service on the target platform it calls the execute function with the parameters to send plus the configuration file.

The execute interface function then does the following:

- it does the "marashalling" (rpc style) in xml of all the quantities declared in adaptation netlist to whose the action belongs to
- it takes the input parameters and re-map them according to the target co-simulation IP block (it acts like a general purpose compiler by mapping local variables into corresponding IP blocks components such as registers, memory blocks, etc ...)
- it loads a cross-compiled obj file if an entire action has been mapped and is ready to be tested
- it sends the whole xml packet to the other side

The receiving pair does the following when it is started. It first call the hw_cosim input interface function call to receive input parameters.

When the xml stream arrives it does the following:

- it first reads the xml stream and does the un-marshalling of the paramters
- it calls the different subsystems to set up IP blocks memory elements correctly
- it updates the contents of hw_quantity_mgr class that provides similar features of metamodel counterpart

Once all the status is set correctly then the IP block is allowed to start execution.

The results and the quantity back-annotation are updated by the IP block itself that implements the hs_adapter class output function which does the following:

- does the "marshalling" of all the annotated quantities
- does the "marshalling" of the results by re-mapping them back into a pre-defined structure that is declared in the configuration file
- sends the stream back waiting for a new request (keeping status information if necessary)

7.3 An example: modelsim

In the modelsim example a simple adder has been implemented. In functional part a for loop executes an "adder" action that is mapped onto an adder realized in VHDL and simulated in modelsim.

The exchanged values use the registries to store accumulator values and return value.

The registry mapping is specified in the xml configuration file both for input registries and for output registry.

In order to co-simulate with modelsim two interfaces require to be implemented:

1. the adapter called hs_adapter
2. the registry access C++ class called hs_registry

The hs_adapter uses the PLI interface of Verilog/VHDL to communicate with the Verilog/VHDL simulator.

Every communication API (input, output) defined in the adapter medium must be static and are registered in the veriusertfs[] structure.

This structure defines the API that are accessible from modelsim models.

This structure realizes the medium modelsim side. As can be seen it is highly IP block dependent but the layer required is very thin. Only few lines of code are required to obtain the whole co-simulation medium adapter for modelsim.

Another important communication interface required to be implemented is the hs_registry.

The two communication functions implemented are set_value and get_value. As their name suggest they set/get the value into register. The registries are accessed by their names, names that are mapped directly in the configuration file. The values are passed as characters strings and converted inside modelsim automatically. It is enough specify accDecStrVal as value_s.format.

7.4 An example: Pipesim ISS

In real world applications where the difference of using a processor or another one may impact a lot in terms of production costs the ability of trying different processors with the easiness of replace and plug one after the other is a very appealing feature.

The pipesim instruction set simulator adapter is an adapter that allows co-simulation between metropolis and any instruction set simulator.

The name pipesim due its origin to the fact that the adapter creates a pipe with the ISS through which it is able to exchange data with it.

Many instruction set simulators are available. A successful prototype for an mp3 player has been done with simplescalar (http://www.simplescalar.com).

When the pipesim adapter is used it acts like a server and waits for connections from metropolis corelib.

When an action is mapped on an ISS the following information are needed in the xml configuration file:

- The remote command line call (sim-safe, sim-outorder, etc)
- The obj file name that implements the action cross-compiled for the target ISS
- The mappings of the memory and the registers (initial conditions).