Interoperability between UML/MARTE Platform Independent Models and Synchronous Data flows in ForSyDe

Microelectronics Engineering Group
of the
University of Cantabria

Authors: F. Herrera, P. Peñil, E. Villar

Date: 2015, April

1 Introduction

The CONTREX UML/MARTE modelling methodology enables the description of the whole multicore MPSoC of a System-of-Systems (considering nodes connected through a network). The modelling methodology enables the building of models which serve as entry point for software synthesis for MPSoC platforms and a simulation-based DSE flows.

A CONTREX UML/MARTE model consists of an application or platform independent model (PIM) and a platform model. Specifically, in this work we focus on the PIM description and on its relation to formally-based modelling methods. In the CONTREX UML/MARTE methodology, the PIM is a component-based model, i.e. the functionality is encapsulated within components, which enables SW reusability. Moreover, the model is annotated with a rich variety of attributes with a standard semantics, encompassed by methodology-based assumptions, which enable to capture many relevant aspects of the PIM semantics.

In this document we tackle the connection of such CONTREX UML/MARTE PIM with formally based modelling methodologies, specifically the ForSyDe methodology. This is very relevant in the context of mixed-criticality systems, where at least some part of the system is tied to strict requirements, not only on performance, but also on predictability. It imposes constraints both on the platform and also at higher levels, i.e. the PIM. In effect, a constraint on the platform might be of the form “do not allow cache memories”, so the analysis of worst-case execution times is feasible. However, as mentioned, the predictability challenge also reaches the application-level, with ever growing complexities, and many potential issues line non-determinism, deadlock, unboundeness, etc, difficult to solve without a disciplined application modelling. Formal-based modelling based on Models-of-Computation (MoC) theory helps on that purpose, since it defines modelling elements, modelling rules and assumptions, which help a modeler to build models enabling correctness-by-construction and more analyzability.

Therefore, this work tackles how the CONTREX UML/MARTE methodology can be restricted in order to satisfy the behavioral and communication restrictions imposed by specific MoCs. This will enable to build models with parts endorsed with such formal ground.

Specifically, in CONTREX we focus on the relation and integration with ForSyDe. ForSyDe support several models of computation and heterogeneous models. Such models can be executed to perform functional validation. Moreover, ForSyDe models are the entry point to an analytical Design Space Exploration methodology and to tools for automatically target GPGPUs and NoC-based implementations (from SR MoC-based models).

Therefore, the interoperability of the modelling methodologies will enable the application and cooperation of their respective tool chains enabled by each modelling methodology. From the CONTREX UML/MARTE perspective, the main benefit has been already mentioned, of having parts of the model formally supported, with well sounded semantics and ready-to-use analysis to guarantee important properties, such as functional determinism, continuity, etc. From the ForSyDe side, the advantage is to have a link with a methodology which can grow the model with more generic parts not tied to MoC rules, and the possibility to exploit the simulation-based performance analysis and software synthesis tool-chains.

1.1 Description of the PIM in the CONTREX UML/MARTE methodology

The purpose of this section is to provide a compact overview of the specification of the Platform Independent Model in the CONTREX UML/MARTE modelling methodology. More specifically, of the PIM model which concerns to the relation with ForSyDe. For it, the overview provides an abstract view, avoiding details referring to the use of UML, and of the MARTE and CONTREX specific profiles. Similarly, the discussion will focus on the MoC concepts supported by ForSyDe, rather on details proper of any of the ForSyDe flavours (in XML, in SystemC and in Haskell).

Figure 1 sketches the type of component based model supported and the primary information which can be annotated and associated to it. The sketch focuses on the most relevant elements, specially concerning to the executive semantics.

The CONTREX PIM model is structured as a set of communicating components. Components communicate among each other by providing and requiring functionalities. Such communication is performed only via component ports. A component which implements a functionality can make it available to another component by encapsulating the function prototype within a *provided client-server interface* and making it public to other components through to *provided port* (i.e., a port with a provided interface). Similarly, components requiring such services will declare a port with matching *required client-server interface*. This way, components communicate among them only through their ports.

A methodological assumption is that all the functions of an interface will have the character of the interface (either provided or required). In other words, an interface cannot be used to contain both provided and required methods.

Additionally, an interface operation can be of any of the following three types:

- *Sequential*: No concurrency management mechanism is associated with the operation and, therefore, concurrency conflicts may occur. Instances that invoke the operation need to coordinate so that only one invocation to a target on any operation occurs at once.
- *Guarded*: Multiple invocations of a behavioural feature may occur simultaneously to one instance, but only one is allowed to commence. The others are blocked until the performance of the currently executing operation is complete. It is the responsibility of the system designer to ensure that deadlocks do not occur due to simultaneous blocks.
- *Parallel*: Multiple invocations of a behavioural feature may occur simultaneously to one instance and all of them may proceed concurrently.

The methodology assumes that all the interface operations are of the same type.

The model supports structural hierarchy once a component instance can be encapsulated within parent components. Eventually, a model could be transformed to a flattened model, semantically equivalent, and where all the components were “leaf components”.

The leaf components are components stereotyped as RtUnits (referred to simply as "RtUnit" in this summary) encapsulate functionality. The model states nothing about the inner structure of the functionality. For instance, both sequential and object-based code can capture the functionality. However, a set of attributes enable to state its due semantics, and how RtUnits interact with other RtUnits via port-to-port communication.

The relevant attributes stating on the RtUnit semantics are the following:

- *isDynamic*:
    - If true, it denotes that the real-time unit creates dynamically the schedulable resource required to execute its services.
    - If false, the real-time unit owns a pool of schedulable resources to execute its services.
- *srPoolSize*: Size of the schedulable resource pool of a real-time unit. It states the capacity of the RtUnit in terms of maximum amount of schedulable resources.
- *srPoolPolicy:* Kind of pool policy adopted by a real-time unit. The possible values in MARTE are:
    - *infiniteWait*: If the pool is empty, the real-time unit waits indefinitely until a schedulable resource will be released.
    - *timedWait*: If the pool is empty, the real-time unit waits for bound time until a schedulable resource will be released. At the end of the waiting time, if no schedulable resource have released, an exception is raised.
    - *Dynamic*: If the pool is empty, the real-time unit creates a new schedulable resource and adds it to the pool.
    - *Exception*: If the pool is empty, the real-time unit raise an exception.
    - Other
- The *isMain* attribute can be set true to denote the main application. Default false.

The methodology enables a rich description of the component communication.

In the simplest case, a simple port-to-port connection can be specified. For such a case, a remote procedure call (RPC) applies. That is, the RtUnit with the required interface (client RtUnit) is the one which calls the *service*, i.e. a function of the required interface, which is implemented and executed by the RtUnit associated to the provided interface port (server RtUnit). The client RtUnit is assumed to be blocked while the server RtUnit receives and process the service requests, and until it returns and notifies its completion to the caller RtUnit.

In order to support additional communication and synchronization mechanisms, the port-to-port connection can be specified as a channel. In such a case, the following attributes serve to modulate the communication semantics:

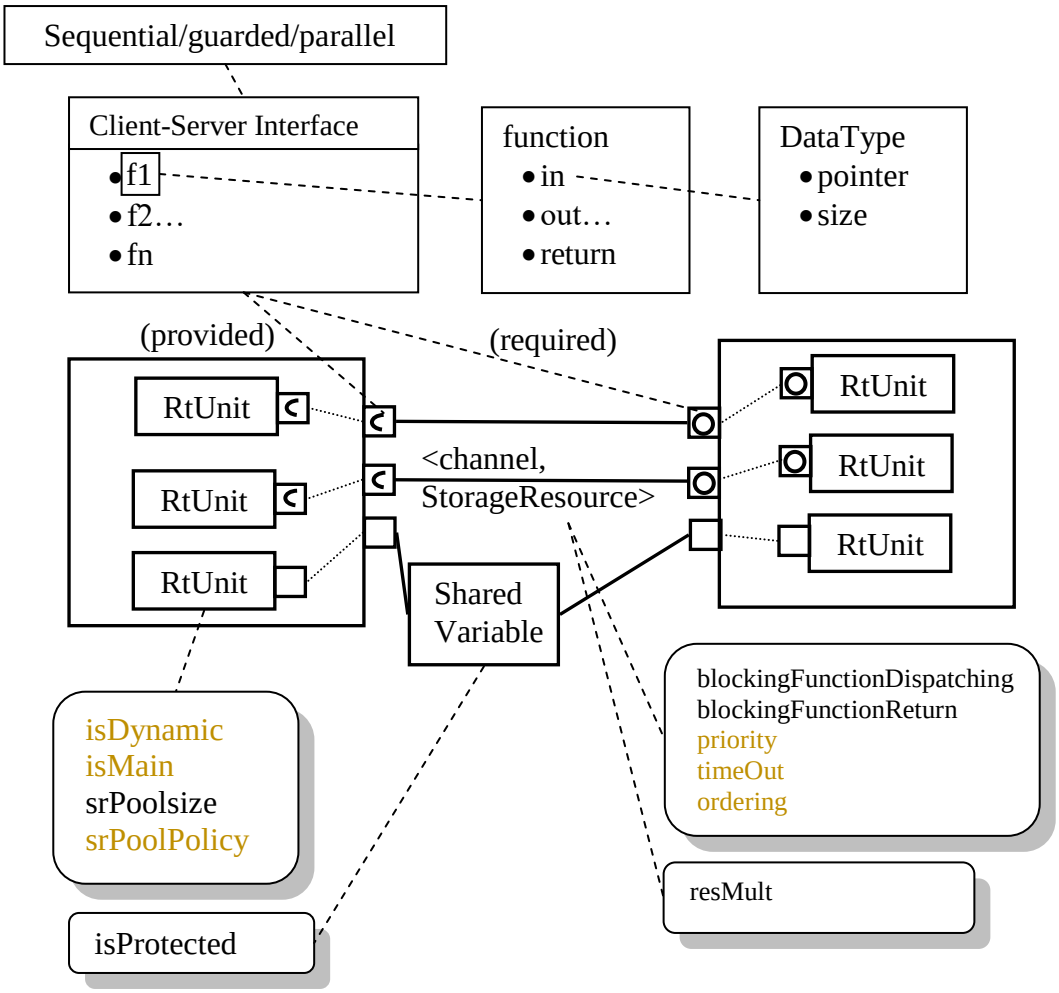


Figure 1 Sketch of the CONTREX UML/MARTE PIM model and the main attributes defining the semantics.

- *blockingFunctionDispatching*: States if the client RtUnit has to remain blocked until either the server starts to attends the service (dispatching) or the call is stored in channel (when the channel has capability at least for a call).

- *blockingFunctionReturn*: States if the client RtUnit has to remain blocked until the server response (including the return values).

- *priority*: states the priority of the client call with regard to the server. Used by the server to decide which client to attend first.

- *timeOut*: maximum time that a client shall be waiting for a function´s call response.

- *Ordering*: states if the channel has to deal with the call as an ordered set and so has to preserve an order in the server response according to the calling order.

In addition, the channel can be also stereotyped such a storage resource capability (result attribute) can be assigned. This storage resource capability states the buffering capability of the channel in terms of maximum amount of function call requests.

The communication semantics associated to the channel is summarized in the Table 1 [1]. Table 1 reflects the communication semantics as a result of four factors. The two former are two static attributes of the channel of the model on the two first columns:

1. Value of the *blockingFunctionDispatching* attribute.
2. Value of the *blockingFunctionReturn* attribute.

The other two columns reflect the state of the channel and of the called RtUnit which implements the called function:

3. Room for a Call in Buffer: States if there is room for storing a function call at a given time during the execution. Notice that this depends on the size of the buffering capability of the channel (*resMult*), among other factors.
4. Schedulable Resource Available for the call: the RtUnit implementing the called function can allocate immediately one schedulable resource (thread) in order to attend to service request. Notice that, among other factors, this depends on the *srPoolSize* attribute value and on the state of the called RtUnit. Specifically, the number of schedulable resources (threads) available must be strictly bigger than the number of pending calls in the buffer, assuming that the pending calls have equal or bigger priority. Then, the RtUnit can use the schedulable resources available for the pending calls and use at least one schedulable resource more for attending the new call.

Therefore, depending on the value of the two first attributes, and on the two state values reflected in columns 3 and 4, the behaviour of the requester (client) RtUnit, of the channel and of the server RtUnit are stated:

5. Store: States if the function call request will be stored or not in the channel at the moment of the call.
6. Block on call: States if the caller (client) RtUnit will be blocked.
7. Block on return: States if the caller (client) RtUnit will be blocked.
8. Exec: States if the called function will be executed or not, and in the former case, if its execution will be delayed until schedulable resources of the called RtUnit are available.

Static properties	**Run-Time State**		**BEHAVIOUR (Semantics)**		
Of channel	**Of channel**	**Of Called RtUnit**	**channel**	**Caller (Client) RtUnit**	**Called (Server) RtUnit**

[1] Notice that Table 1 covers part of the attributes. Considering all the attributes and their possible values (complete semantics description) would lead to a high amount of combinations. The relation of the methodology to formally based methodologies enables focusing on combinations which lead to interesting properties.

	Blocking Function Dispatching	Blocking Function Return	Room for a Call	Sched Res. Available	Call Stored	Block on Call	Block on Return	Exec
	1	2	3	4	5	6	7	8
1	true	true	Yes	Yes	No	No	Yes	Yes
2	true	true	Yes	No	Yes	No	Yes	Delayed
3	false	true	Yes	Yes	No	No	Yes	Yes
4	false	true	Yes	No	Yes	No	Yes	Delayed
5	false	false	Yes	Yes	No	No	No	Yes
6	false	false	Yes	No	Yes	No	No	Delayed
7	true	false	Yes	Yes	No	No	No	Yes
8	true	false	Yes	No	Yes	No	No	Delayed
9	true	true	No	Yes	No	No	Yes	Yes
10	true	true	No	No	No	Yes	Yes	Delayed
11	false	true	No	Yes	No	No	Yes	Yes
12	false	true	No	No	No	No	No (*)	No
13	false	false	No	Yes	No	No	No	Yes
14	false	false	No	No	No	No	No	No
15	true	false	No	Yes	No	No	No	Yes
16	true	false	No	No	No	Yes	No	Delayed

Table 1 Communication semantics to be implemented

The table shows that:

- The calling RtUnit gets blocked ("Yes" in Column 6 for cases 10 and 16) only if the channel is stated in the model as "*blockingFunctiondispatching=true*" and neither there is space in the calls buffer, nor there is schedulable resources available for the call at the moment in time it is done (cases 10 and 16). Then, the execution is delayed until the call is attended. That is, until either, there is space in the call buffer or until there is schedulable resource available to attend the call.
- The call is stored in the channel ("Yes" in Column 5 for cases 2,4, 6 and 8) only if there is room in the call buffer at the moment of the call and the RtUnit cannot allocate immediately a schedulable resource (thread). That happens when the number of Schedulable Resources Available in the called RtUnit is bigger than the number of pending calls in the channel buffer.
- The called function is always executed except when when, at the time of the call, neither there is room to store the call in the call buffer, nor there is a schedulable resource available to attend the call and the channel was stated in the model as "*blockingFunctiondispatching=false*" ("No" in Column 8 for cases 12 and 14). In those cases, the function call is lost and can never be attended.
- The called function will be executed immediately (marked as "Yes" in Column 8 for cases 1,3,5,7,9,11,13 and 15) whenever there is a schedulable resource available at the time of the call.
- The called function will be executed after a delay (marked as "delayed" in Column 8 for cases 2,4,8,10 and 16) if:
    - (a) there was no schedulable resource available, but there was room in the call buffer to store the call (cases 1,3,5 and 7), or
    - (b) there was neither room in the call buffer nor schedulable resource available, but the channel is stated in the model as "*blockingFunctiondispatching=true*".

- In general, the caller function is always blocked waiting for the return whenever the "*blockingFunctionReturn=true*" is stated. The exception is reflected as "No (*)" in column 7 of case 12, when the model states "*blockingFunctionReturn=true*" but the semantics of the return is not blocking. The reason is that for this case, where a call is done with "*blockingFunctiondispatching=false*", and where at the time of the call there is neither room on the call buffer, nor available schedulable resource to attend the call immediately, it that it means that the call will be lost. Therefore, a blocking wait would mean a deadlock, since the call will never be attended. For this case it is assumed that the channel will detect the situation and it will immediately return with an error condition. Therefore, it can be seen as a blocking on a return value which will be immediately returned, so as no blocking.

Finally, there is a third communication mechanism supported, based on shared variable. For it, the connector is stereotyped as a shared variable, which enables two attributes:

- isProtected: States if the client RtUnit has to remain blocked until the server starts to attends the service (dispatching).

- type: type of data of the shared variable.

A CONTREX UML/MARTE PIM supports some additional features which are relevant for the later relation with the ForSyDe (SDF) models. A first feature of interest is the possibility of specifying a *joint service*. It is sketched in Figure 2.

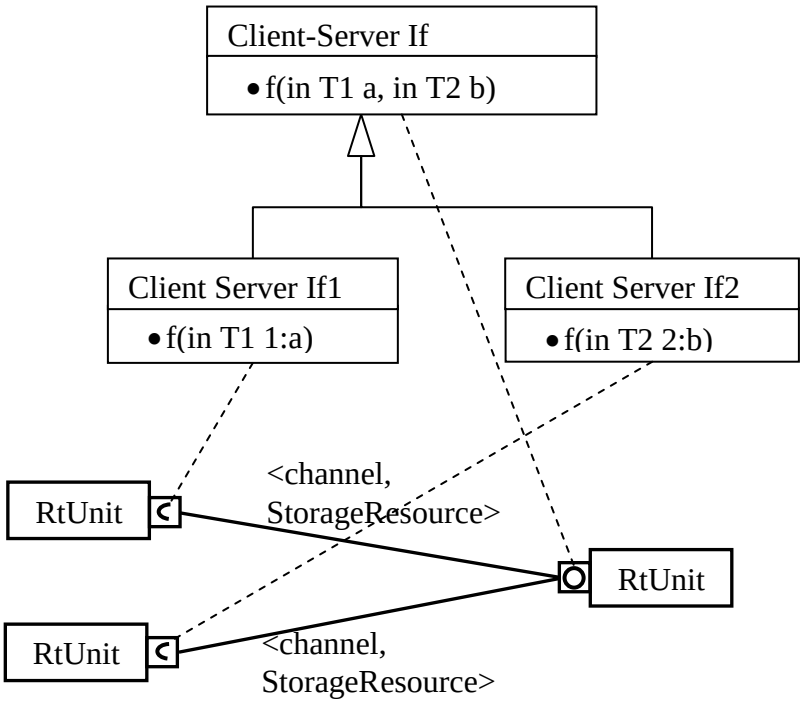


Figure 2 Sketch of a joint-service in the CONTREX UML/MARTE PIM modelling methodology.

The basic idea is that a RtUnit can provide a service implementing a functionality "f(in a:T1, in b:T2)" and that such a service can be invoked from two different RtUnits providing independently the input arguments.
A methodological assumption is that the channel confluence in the provided port means such joint service and that the serviced will not be dispatched until all the input arguments are available.

In order to support this modelling pattern, the methodology assumes that the interfaces "If","If1" and "If2" are compatible as long as "If" is the provided interface, and the "If1" and "If2" interfaces are required interfaces, and have been defined as specialized interfaces inheriting from "If" (as shown in Figure 2).

Another feature supported by the CONTREX UML/MARTE modelling methodology is "*data-splitting*". Data splitting is oriented to model, explore and exploit data parallelism.

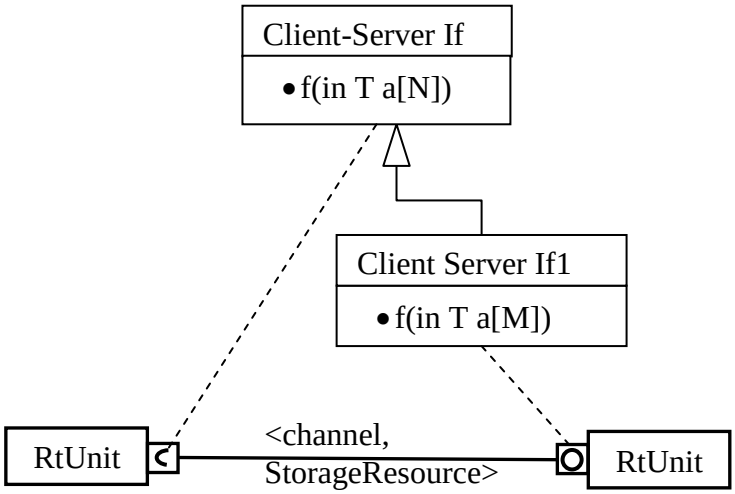


Figure 3 Sketch of data-splitting in the CONTREX UML/MARTE PIM modelling methodology.

Supporting data-splitting means that the interface of a provided service "f" can be declared such the caller execution processes an amount of data M of a given type "T", which can be lesser than the amount of data N of the same type "T" contributed at the call of the service, that is M<N. The methodology understands that, as long as, the interface "If1" is declared as a specialization of the "If1" interface, they are compatible and the "data-splitting pattern" is inferred.

Data splitting means that the server RtUnit will have either to fire a number of times "a" in a sequential manner or by to using "b" (with b≤"a") schedulable resources in order to process all the M data at the service request. Moreover, the data returned from the split calls have to be joinable and joined.

The number "a" ranges between $\lfloor N/M \rfloor$ and $\lceil N/M \rceil$ times depending on whether previous calls have left remaining input arguments from previous service calls. This is possible, since the methodology does not obliges M to be an integer multiple of N.

A generalization of the "data-splitting" modelling pattern means to support the case M>N. This is supported by involving several firings of the requesting RtUnits before executing the service.

Moreover, the CONTREX modelling methodology covers a more generic pattern of joint-service, which, as it is sketched in Figure 5, supporting joint-services with data splitting.

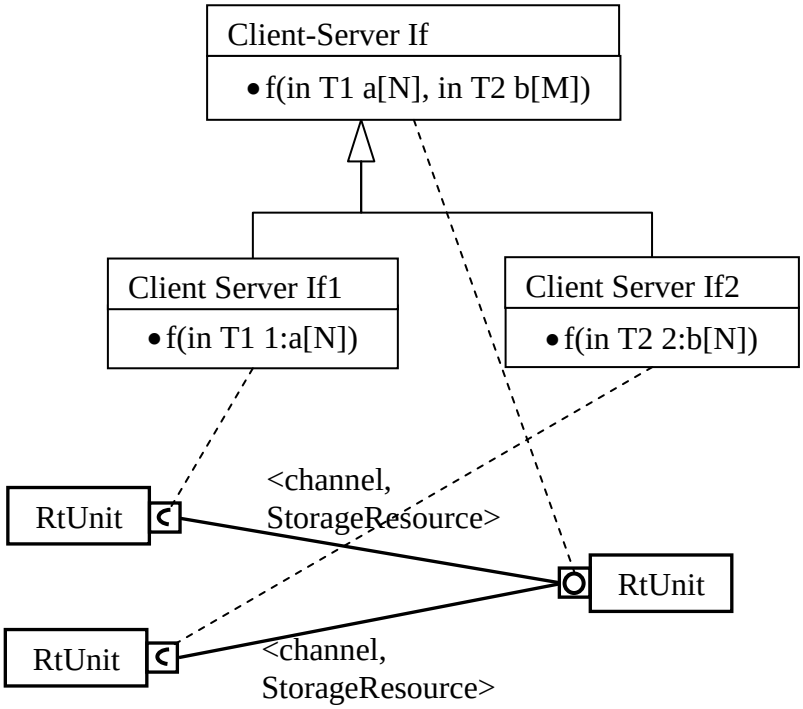


Figure 4 Sketch of a joint-service with data-splitting in the CONTREX UML/MARTE PIM modelling methodology.

Nota: Pablo, ¿es este el patron al que te refieres? Lo que me parece curioso es que en el de joint service la interfaz provista es la interfaz padre, mientras que en el data splitting la interfaz provista es la interfaz hija. Asi que, ¿Cómo se hace el “merging”? No me queda claro.

2 Importing ForSyDe SDF Models into the CONTREX UML/MARTE PIM

This section shows how a ForSyDe SDF model is imported as a CONTREX PIM model. There are several patterns which can be used to import the SDF model. This section shows a pattern which has been selected because:

- The pattern is supported by an existing software synthesis methodology
- It reflects a natural modelling of SDF relying in function-based communication, where the sense of the calls is the same that the sense of the flow of the data

Section **REF** discusses the patterns that have been found so far and how they can be also employed in the MARTE to ForSyDe export.

2.1 Import a simple directed Homogeneous SDF model

In order to reason on the relation between the CONTREX UML/MARTE methodology and ForSyDe SDF untimed model, a first convenient step is to reason on a simple SDF pattern, specifically on a simple homogeneous SDF graph, shown in Figure 5.

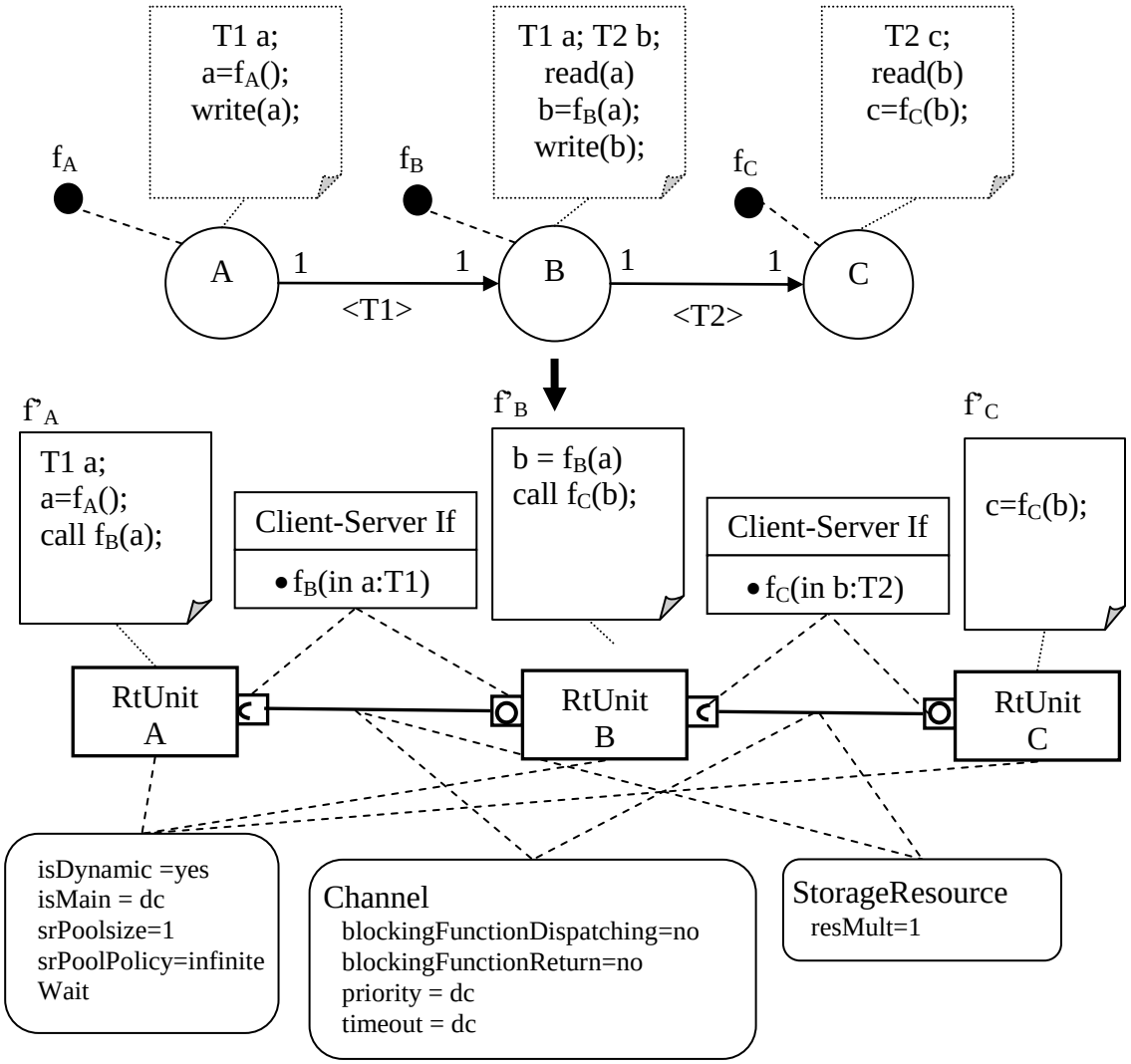


Figure 5 . ForSyDe to CONTREX MARTE model mapping of a simple homogeneous SDF graph.

In an SDF model, a generic SDF node (1) first reads input data, (2) then computes, and (3) then sends output data (e.g., node B in Figure 5). There are two specific derived types of nodes: source nodes and sink nodes. Source nodes have no reading inputs, so they are not waiting for data and the can autonomously trigger, i.e. they are autonomous nodes. The have associated functionality without input arguments capable to generate output data (e.g., in Figure 5, f_A in node A). In contrast, all nodes with inputs are reactive, in the sense that they remain blocked until there are sufficient data in their input to be fired. The sink nodes are a specific kind of reactive node which produces not ouput data for another SDF node (e.g., node C in Figure 5). Notice that when a SDF model is specified, the internal structure of the node is directly associated to its type, and so it is sufficient to synthetically represent the association of the function to the node (e.g., of function f_A to node A, of function f_B to node B, of function f_C to node C in Figure 5).

On the bottom of Figure 5, a sketch of the CONTREX UML/MARTE model derived is shown. Relying on that sketch, a fundamental set of mapping rules is derived:

1. Each SDF node (origin[2]) is mapped into a RtUnit component plus an associated functional code (target[3]). The attributes of the target RtUnit must fulfil:
    a. A source HSDF node is mapped to a RtUnit with "isMain=true".
    b. The remaining nodes (generic or sink) will have "isMain=false".
2. The code associated to a RtUnit targeted from a generic node must fulfill the following rules:
    a. It must implement a provided interface function f'(T) with the same interface and corresponding to the function executed by the origin single-node HSDF node f(T), where T is the data type of the origin input edge.
    b. f'(T) must contain an internal variable for each return/output value produced by the execution of the provided interface function.
    c. f'(T) must contain at the end a call to the interface function corresponding to the functionality of the destination single-SDF node, using as parameter the internal variable mentioned in Rule 2.b.
3. The code associated to a RtUnit targeted from an origin source node must fulfil rules 2.a, 2.b and 2.c, with the particularity that the origin function has no parameters. Then f'(T) is triggered autonomously (recalling Rule 1.a).
4. The code associated to a RtUnit targeted from an origin sink node is a particular case of Rule 2, where only rule 2.a applies.
5. Every *srPoolPolicy* attribute of a RtUnit component with provided ports has to have the value *inifiteWait*.

[2] We use *origin* adjective for elements on the origin model (ForSyDe model in the ForSyDe to UML/MARTE mapping)
[3] We use *target* adjective for elements on the target model (UML/MARTE model in the ForSyDe to UML/MARTE mapping)

6. Each SDF edge is mapped to a port-to-port link. The origin of the edge is mapped to a required port. The end of the edge is mapped to a provided port. Both ports refer to an interface with a prototype of the function implemented by the server RtUnit, which in turn corresponds to the functionality of destination SDF node (node reached by the SDF edge).

7. The port-to-port link shall have an associated channel stereotype with the following values:

    a. blockingFunctionDispatching=false

    b. blockingFunctionReturn=false

Summing up, these rules support a dataflow within a component-based modelling methodology with a generic and flexible function-call based communication (i.e. the the CONTREX UML/MARTE methodology). It consists of autonomous (RtUnits) which call and trigger services provided by other (RtUnits). Function calls are asynchronous in the dispatching and in the return the caller RtUnit do not mind the return value. Actually, the output values of the called functions are used by the owner RtUnit to trigger other RtUnits corresponding to the triggered origin SDF nodes.

It is worth to make some clarifications on the rules.

Notice that Rule 1 maps to a RtUnit, regardless the type of the origin SDF node. The consideration of the type of SDF node is translated into a specific constraint on the code associated to the target RtUnit component. The “isMain” attribute is used to distinguish autonomous from reactive functionality.

Rule 2 means that in the most generic case the functionality associated to the mapped RtUnit must fulfill some constraints, that is, a structure which requires the generation of a wrapper function. For instance, the SDF node B is converted into a RtUnit plus an associated functionality $f'_B()$ with exactly the same input arguments as functionality f_B. Then, the caller RtUnit A actually invokes the wrapper function f'_B, which, in addition toinvoking f_B, uses the produced data to invoke the function the corresponding wrapping function f'_C, in charge of executing the functionality corresponding to SDF node C.

Rule 3 means that RtUnit A will contain f_A functionality (without arguments), which autonomously triggers.

Rule 4 in Figure 5 means that f_A functionality must be executed in RtUnit C. Notice that dumping the output/return data on an internal variable and making something with it is not strictly required from the translation perspective.

Rational on the Rule 5 is that such attribute guarantees that there will not be data lost in the communication (a key assumption in dataflow models) because there will not be any function call service lost.

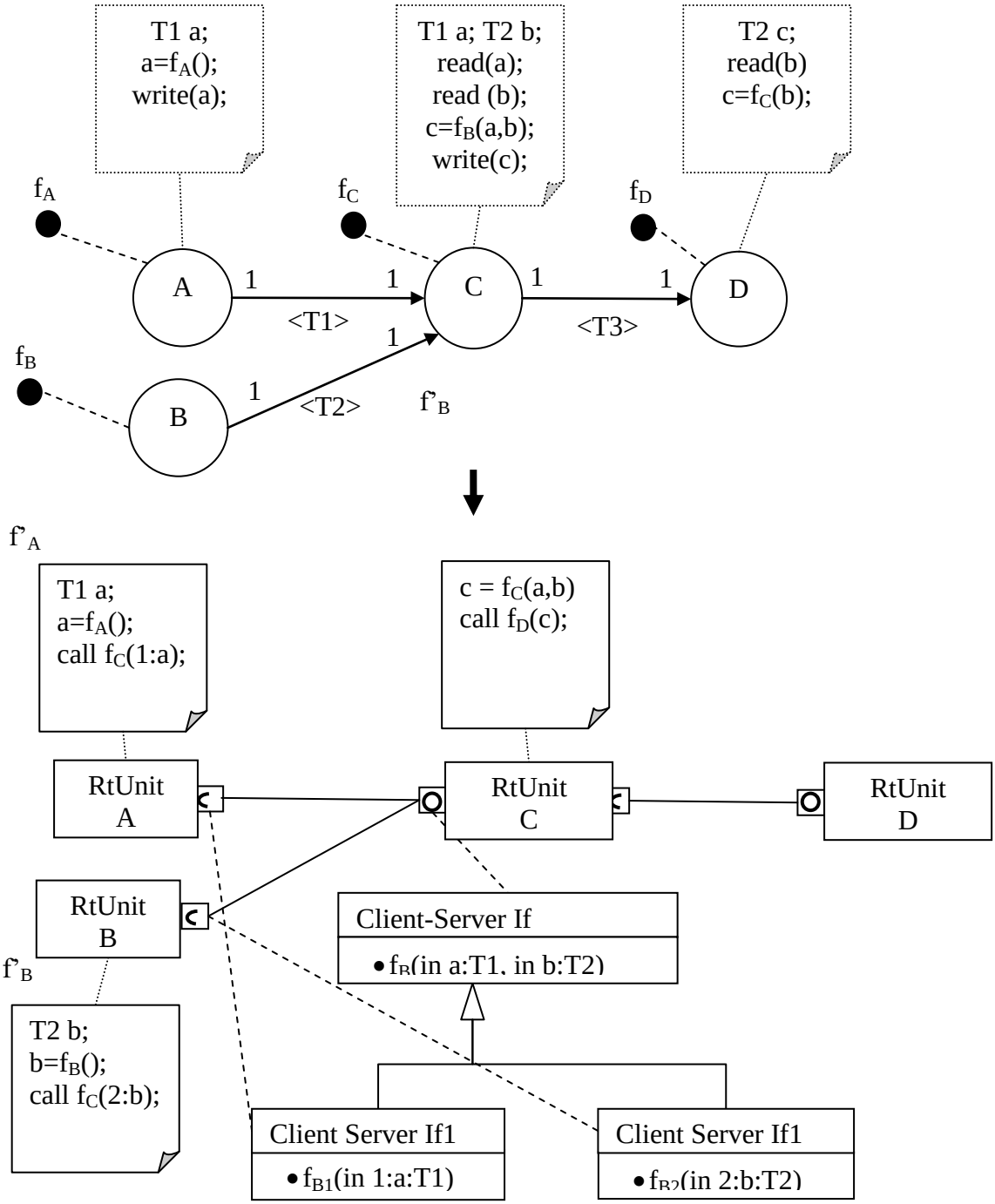
Other parameters such as *idDynamic*, *srPoolSize* and ordering are implementation details are not relevant and can be considered don't care (dc in Figure 5). In order to guarantee that the imported model is PHARAON compliant (so to reuse SW synthesis tooling in its current state), the following attribute values can be assigned by default:

isDynamic=true

srPoolSize>0, e.g. srPoolSize

ordering=true

The association of a *StorageResource* parameter for modelling is similarly don´t care. However, it is an implementation detail which is not reflected in the original SDF model. A refinement of the model could consider that there should be a minimum storage capacity for an untimed implementation (relying on blocking communications and dynamic scheduling). For the HSDF case the value 1 is used.


T1 a;
a=fA();
write(a);
T1 a; T2 b;
read(a);
read (b);
c=fB(a,b);
write(c);
T2 c;
read(b)
c=fC(b);
fA
fC
fD
A
C
D
1
<T1>
<T3>
fB
B
<T2>
f'B
f'A
T1 a;
a=fA();
call fC(1:a);
c = fC(a,b)
call fD(c);
RtUnit
A
RtUnit
C
RtUnit
D
RtUnit
B
Client-Server If
• fB(in a:T1, in b:T2)
T2 b;
b=fB();
call fC(2:b);
Client Server If1
• fB1(in 1:a:T1)
Client Server If1
• fB2(in 2:b:T2)