COMPONENT BASED APPROACH FOR SDR WAVEFORM DEVELOPMENT ON DSP TARGETS

Laurent Poyart (THALES Communications S.A., Colombes, France; laurent.poyart@fr.thalesgroup.com); Thomas Derive (THALES Communications S.A., Massy, France; thomas.derive@thalesgroup.com); Eric Nicollet (THALES Communications S.A., Colombes, France; eric.nicollet@fr.thalesgroup.com)

ABSTRACT

The proliferation of waveforms and SDR platforms coupled with their increasing complexity implies the need of a structured software design to gain in modularity, flexibility, reuse and portability.

This paper describes a global approach using Model Driven Engineering (MDE) and Component Based Software Engineering (CBSE) for waveform design on SDR platforms with heterogeneous processing units (DSP, GPP, FPGA) and particularly focuses on the DSP solutions proposed to unify the approach initiated on the GPP side.

The paper will bring out the benefit of the component-based approach to automatically adapt whole or part of a SDR waveform from an Operating Environment (OE) to another with different constraints and characteristics. This capability is one of the most important key elements for future SDR.

The use of a tool-aided framework to define and deploy generic software components as SCA resources over CORBA (compliance to SCA 2.2.2 [1]) or an other specific middleware (SCA next “CORBA optional” orientation) will be addressed. The article will particularly focus on DSP concerns and some performances and portability results will be provided to illustrate the benefits of the approach.

The work presented in this paper has been done thanks to the European Commission funding for the EULER project [5] of the Framework Programme Seven, Cooperation, Securities theme, Grant Agreement FP7-SEC-218133.

1. MDE AND CBSE

Model Driven Engineering refers to a range of development approaches based on the use of software modeling as a primary form of expression. Information contained in the model is used to transform design to code and test artifacts. MDE involves automatic model transformation which plays a critical role since it automates complex, error-prone, and recurrent software tasks. Combined with Component Based Software Engineering, it promotes a better structure of software with separation of concerns between infrastructure and business logic.

Component based software relies on assemblies of interconnected components. A component is a unit of composition with specified interfaces (required and provided). The interfaces represent the contracts between components.

Combining MDE and CBSE greatly improves embedded software development portability and productivity.

2. CONTEXT

A crucial issue proposed by SDR programs today is to be able to deploy a same waveform on several SDR platforms. The SCA gives reference architecture answers to face this challenge. Nevertheless, the SCA specification separates CORBA and non-CORBA processing units and gives some architecture guidelines which doesn’t address those processing units on the same level. On one hand, the SCA describes components named Resources for the CORBA processing units while the non-CORBA processing units are addressed at the communication level with the MHAL extensions (MHAL Device, MHAL Comm [6]). Moreover, the use of those extensions impacts the specification of the GPP SCA Resources which need to mix functional ports with non-functional MHAL ports, limiting the portability of those components on heterogeneous middleware solutions.

In the SDR EULER project [5], it was needed to face to portability constraints, particularly concerning the DSP. Two kinds of SDR platforms were to be addressed: one platform based on the CORBA middleware on the GPP and the DSP and another platform without CORBA support on the DSP. The initial idea was to unify the approach between the GPP and the DSP and to minimize the porting efforts.

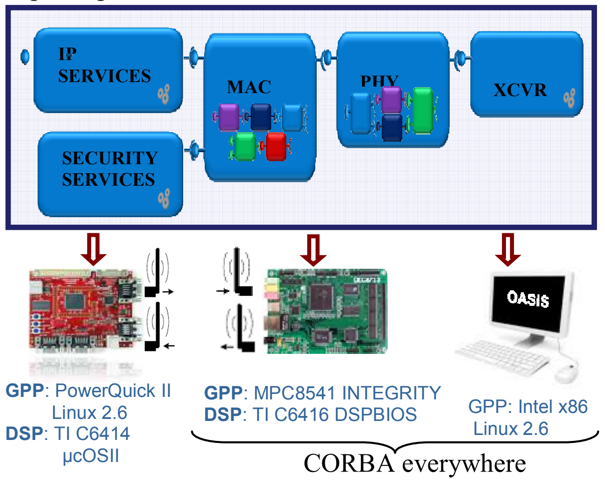


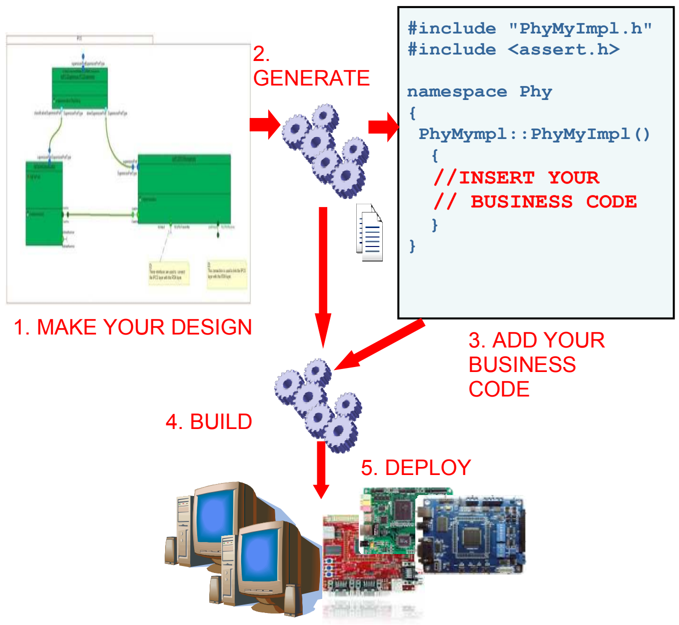
Figure 1 : EULER portability issues

A MDE/CSBE approach was previously used to address SCA architecture on the GPP and the Euler portability requirements were the opportunity to extend it to the DSP and to unify the approach while keeping in mind the specific real-time constraints imposed by the DSP (maximal reduction of the framework footprint and CPU usage).

3. LWCCM FRAMEWORK AND ASSOCIATED DEVELOPMENT PROCESS

For many years, a CBSE approach based on the LwCCM OMG standard [2] has been introduced on GPP to achieve Software Defined Radio designs. This approach is precisely based on the MyCCM (Make Your Component Container Model) component framework which allows defining CCM components using IDL3/IDL. This framework has been extended to support the SCA Resources constructed by an assembly of CCM components and deployed using a Core Framework.

The following diagram shows the development process used with MyCCM.



```
#include "PhyMyImpl.h"
#include <assert.h>

namespace Phy
{
 PhyMympl::PhyMyImpl()
  {
   //INSERT YOUR
   // BUSINESS CODE
  }
}
```

Figure 2 : MyCCM development process

The development process follows a top-down approach which covers the development cycle from modeling to tests and integration on the target.

3.1 Make Your Design

In the first step, the components specification is achieved with your preferred modeler e.g. Rhapsody [9], Spectra CX [10] (scheduled). This step allows describing the static structure of the components and their dynamic interactions: composition and collaboration features are specified in parallel. Therefore, at this level, some domain specific constraints and real-time properties can be expressed. Such properties can be threading properties, resource usage constraints or any kind of properties which can be exploited to configure the model transformation process. The component connections are specified and the CCM components can be grouped to form SCA resources [11]. The MyCCM framework offers also the possibility to specify the components and their deployment in conformity with OMG D&C specification [4]. The components need to be defined in IDL3, the interfaces in IDL and their deployment in a Component Deployment Plan (CDP) XML file. This file contains mainly the components implementations, their instances mapped on the platform and the connections between components.

3.2 Generation

The second step is the transformation step in which artifacts are generated. Those artifacts contain stub/skeleton elements needed in order to manage the cross-connectivity, elements needed for local communications, threading code, CCM containers with respect to the LwCCM containers and finally the SCA containers. The transformation process could also be used to generate mirror components which may serve as test components. Finally the framework generates an implementation template. This template is used by the waveform developer to insert its own business code. The containers ensure a separation between business code and technical code which will favor reuse and also portability improving quick adaptation to various specific platforms.

3.3 WF development

This step is dedicated to the waveform business code implementation. The waveform developers produce the business code which is inserted in the generated component containers.

3.4 Build and Deployment

During the last two steps, the containers are built together with the business code and the deployment is achieved on the target using the deployment files generated by the framework (SCA XML files like SAD, SCD, etc.).

In contexts others than Software Defined Radio, the component framework generators also fit the target specific requirements. Thus, in aerospace domain, ADA code can be generated or also JAVA code for less constrained but more dynamic targeted execution environments. The domain specific adaptations are efficiently managed by generation tools.

4. GLOBAL GPP-DSP APPROACH

4.1 IDL on DSP

One of the key elements of the unified approach between GPP and DSP relies on the use of IDL in the component specification. One of the issues was to define a limited IDL profile in order to reduce its footprint on the DSP side. This profile is described in 5.1.

4.2 CORBA-MHAL bridging solution

For some waveform developments, CORBA is not suitable on DSP side. In this case, the SCA 2.2.2 provides extensions for the so-called "HW processors". The communications between processing units are addressed through the MHAL Comm specification [6]. In order to be able to specify components in the same way on DSP and GPP the operating environment has been enriched to provide an alternative to CORBA. This solution implements a broker pattern with simplified assumptions compared to CORBA. Consequently, the protocol messages are reduced in size and the broker memory footprint is satisfactory.

Some software artifacts (stubs, skeletons, …) have been defined in order to be able to build SCA components based on this dual architecture. The bridging between the CORBA and non-CORBA sides is ensured by deploying an extra component called "proxy". The goal of the proxy is to mirror each DSP based resource on the GPP. This component, implemented as a SCA Resource, is deployed by the Core Framework and connected to other GPP Resources or to other proxies (depending on the deployment plan). The proxy is connected to other SCA Resources through functional ports (in contrast to MHAL ports). Another function of this component is to relay the Core Framework requests (of CF::Resource and CF::Port interfaces) on the DSP side in order to establish, local links on DSP when proxies are inter-connected on the GPP, or GPP-DSP links when the Core Framework connect a GPP Resource to a proxy (mirroring a DSP resource). This function is ensured by using services of the broker. Finally, the proxy performs the CORBA versus non-CORBA transformations for all the functional interfaces of the component. The proxy doesn't contain any business code and can be assimilated as a pure container. The containers for both sides are generated from a single component description. The following figure illustrates this typical architecture implemented during the EULER project in order to fit the CORBA-MHAL bridging solution previously described.

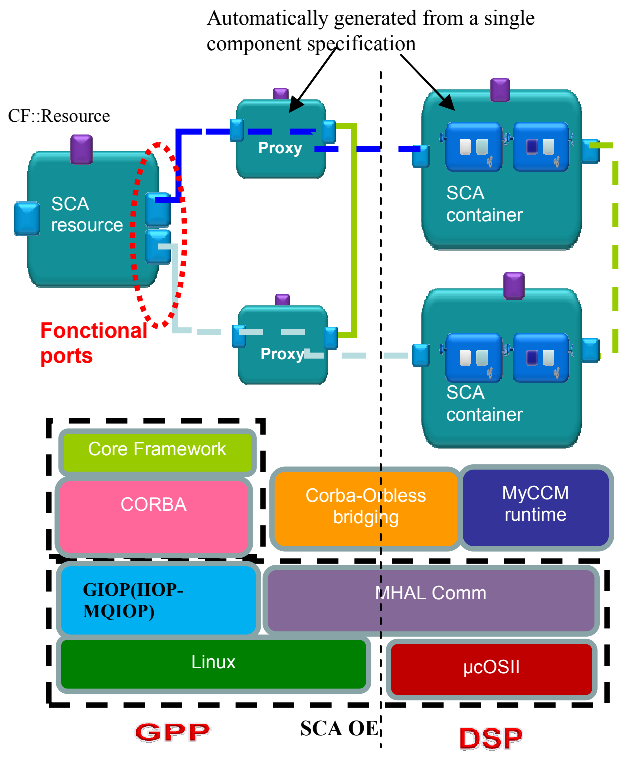


Figure 3: CORBA / non-CORBA bridging

4.3 CORBA everywhere solution

When using a full CORBA solution, the proxies identified in the previous solution are not required anymore because the middleware solution is uniform between GPP and DSP. The framework has been supporting C++ implementations of CORBA middleware for a long time on GPP targets. During the Euler project, the framework has just been adapted to support the PrismTech Openfusion eORB C [7] for the DSP target. No business code adaptations where needed on DSP when moving from previous CORBA-less C++ solution to the CORBA C solution.

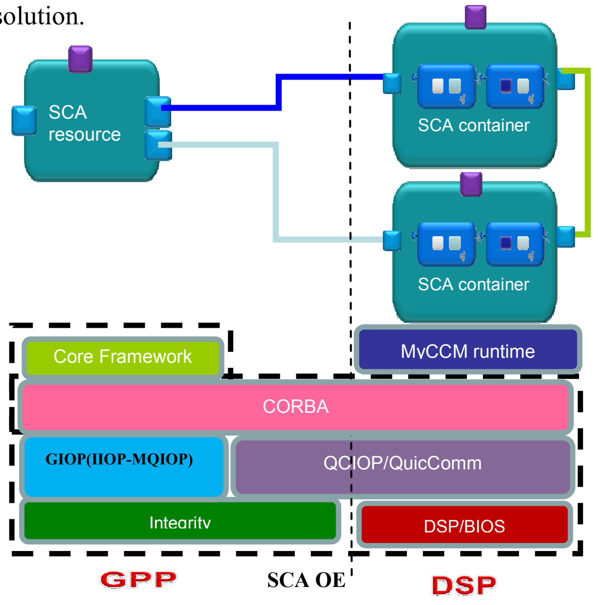


Figure 4: CORBA everywhere

Prior to the integration of the waveform on the CORBA everywhere GPP/DSP platform, the waveform was generated for a full CORBA simulation environment on a Linux host. The migration from one CORBA solution to another is supported by the framework generators. The use of the simulation environment allowed performing native tests offering more debug capabilities and the possibility to validate functional behaviors before the HW availability. To adapt to the simulation environment, it is needed to simulate some platform components e.g. transceiver and to modify business code only if it relies on target specific services e.g. signal processing optimized library.

5. DSP ADAPTATIONS

The development of waveform on DSP has to face more constraints than the GPP, particularly concerning the small memory availability. In order to meet the DSP constraints, the MyCCM framework has been adapted. Some optimizations have been achieved to reduce its memory footprint and CPU usage on the DSP and some extensions have been introduced.

5.1 IDL profile

One of the issues was to define a limited IDL profile in order to reduce its footprint on the DSP side. This profile supports most of the CORBA basic types (char, short, long, boolean, octet, string), union, structures and sequences. Complex types have been removed or are partially supported in the interface definitions in order to support the syntax of the SCA components interfaces (CF::Resource and CF::Port): the *Any* syntax is supported but limited to a few basic types (short, char, long, octet, boolean) in runtime. The *Object* keyword is syntactically supported but doesn't carry any CORBA object.
A specific mapping to C++ has been defined in order to avoid some memory overheads proposed by the default mapping. Component migration from DSP to GPP supported by this profile is really fastened and improved. In order to be able to perform the migration from GPP to DSP, the component specification has to be reduced to the most constrained profile.
This IDL profile is not set rigidly and will surely evolve with new targets introduction e.g. floating point DSPs.

5.2 Real Time patterns

The extensions performed to the framework concern capabilities to handle non-functional properties including real-time properties such as threading, resource usage. The objective of those extensions was to provide some real-time features and particularly capabilities to configure components interactions. This issue is crucial in real-time embedded development and must be available in the first steps of the development process in order to provide to the generators the necessary information from which very efficient code can be produced.

5.2.1 Threading

The first interaction pattern introduced in the MyCCM framework concerns threading capabilities. The framework has been enriched with the well-known Active Object pattern [3]. The framework gives the ability to express threading properties (priority, stack size, queue size) and to affect a thread to one or more component ports (active ports) distributed on one or several components. The design pattern decouples methods executions from methods invocations that reside in their own thread of control. It provides a solution to concurrency. The implementation of the pattern relies only on POSIX API and can be easily ported to several operating environments provided that POSIX AEP profile is available.

The following figure shows an example of threads allocation to components ports:

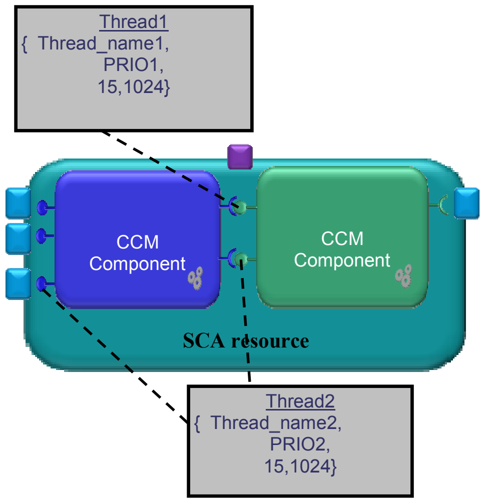


Figure 5: Threading Pattern (active object)

The next figure provides a simplified view of the design pattern.

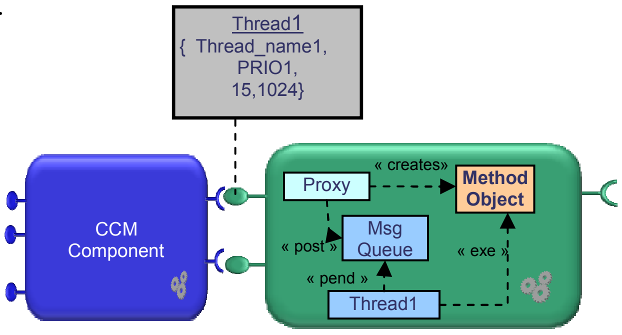


Figure 6: Threading Pattern (active object)

5.2.2 Memory tuning capabilities

Another pattern has been introduced in order to configure resource usage during component interactions and particularly the memory allocations. This pattern concerns essentially distributed interactions and co-localized interactions with an active port. During those interactions, some software artifacts need to be created (method objects for example). The way those objects are allocated has an impact on the interaction performances. It depends on the allocation time which can vary depending on the kind of allocator used. The allocation time, when using the global heap (malloc/new), is well known to vary depending on the number of allocated blocks and the heap fragmentation. Some alternative allocators (local heap, memory partition) are provided with the framework. Those allocators allow to act on either determinism or memory footprint or must be tuned depending on the interaction constraints. The framework gives the possibility to declare memory allocators in the deployment model and to bind them to connections/links between components.

The memory tuning capabilities provide flexibility and control for both determinism and memory consumption during component interactions.

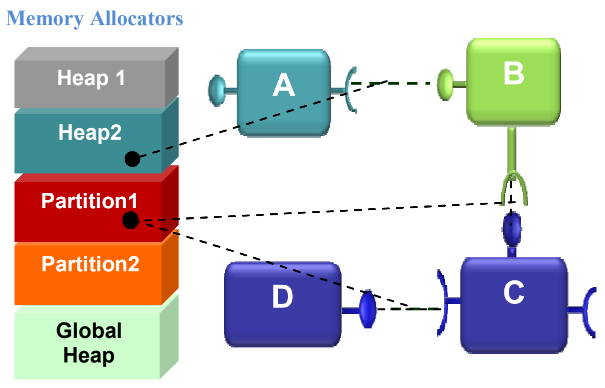


Figure 7: Links configuration

6. BENCHMARKS

Benchmarks have been carried out on a Texas C6416 DSP with 600Mhz CPU frequency and 1Mbytes of internal memory. The figures provided hereafter address the non-CORBA solution on the DSP.

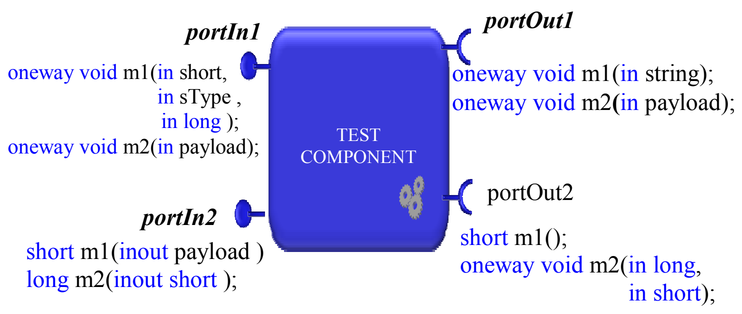
6.1 Memory footprint

The embedded framework memory footprint is small. It represents less than 5% of the internal memory of the DSP. The occupation ratio includes the following elements: the MyCCM runtime, the support to SCA components on DSP, the broker pattern to address cross connectivity, the MHAL Comm and finally a POSIX subset.
Those figures don't include the RTOS and BSP memory footprint.

The next table gives the memory footprint of a reference component. The figures are provided in 3 configurations:

- MyCCM component without thread
- MyCCM component with one thread mapped on input ports
- MyCCM component threaded and with the SCA envelope generation including the cross connectivity artifacts (stubs/skeletons). The SCA component is generated with the same ports as the CCM component and enriched with the CF::Resource interface.

The component used to measure the footprint has the following structure:



```
portIn1:
oneway void m1(in short,
               in sType ,
               in long );
oneway void m2(in payload);

portOut1:
oneway void m1(in string);
oneway void m2(in payload);

portIn2:
short m1(inout payload )
long m2(inout short );

portOut2:
short m1();
oneway void m2(in long,
               in short);
```

Figure 8: reference component

The IDL types used as parameters are:

```
typedef sequence<char,1024> payload;
typedef sequence<short,10> small_payload;
struct sType{
            short _p1;
            long p2;
            small_payload p3;
};
```

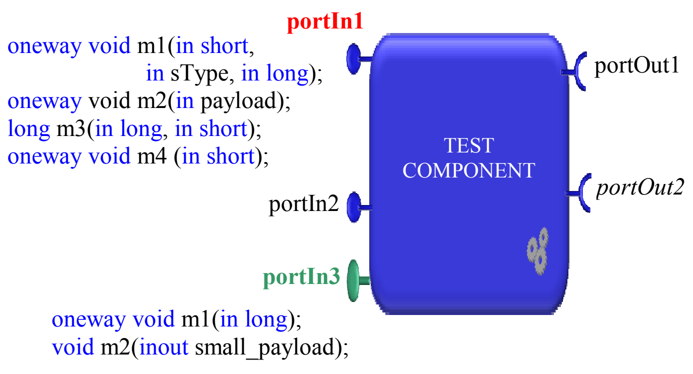
Component Container	Size (Kbytes)
a. LwCCM no thread	1,7
b. LwCCM threaded	4,5*
c. LwCCM threaded + SCA:	18,1*
- SCA resource	5,1
- Cross Connectivity artifacts	8,5

Δa = 2,8K
Δb = 13,6K

Table 1: reference component

*This size doesn't include the thread stack size.

The next table gives the memory footprint of a same component enriched with one port and some new operations on portIn1:



```
portIn1:
oneway void m1(in short,
               in sType, in long);
oneway void m2(in payload);
long m3(in long, in short);
oneway void m4 (in short);

portIn3:
oneway void m1(in long);
void m2(inout small_payload);
```

Figure 9: reference component enriched

It highlights some footprint variations:

Component Container	Size (Kbytes)	
a. LwCCM no thread	1,8	
b. LwCCM threaded	6,4	Δa = 4,6K
c. LwCCM threaded + SCA:	22,3	Δb = 15,9K
- SCA resource	6,2	
- Cross Connectivity artifacts	9,7	

Table 2: reference component enriched

The memory footprint tables show that the LwCCM container for a passive component is small (less than 1%) and grows lightly when expanding the component specification. When the same component is declared with active ports the container size grows in a faster way because it manages the threads communications (synchronizations, parameters copies if needed). The impact of a specific method in the global container size depends on its parameters complexity. Nevertheless it can be observed that the final version of the component with threading management is around 6Kbytes which is less than 1% of the global internal memory.

The tables also shows that the SCA part of the container which includes the SCA component (support of CF::Resource, CF::Port between GPP & DSP, cross connections between DSP resource and proxy) and all the cross connectivity artifacts is about 13,5Kbytes in its first version and 15,9Kbytes with the extended component. This container includes stubs/skeletons and the marshaling/un-marshaling features which were traditionally written by the waveform developers and which are now automatically generated. Classically, in order to avoid too much overhead, DSP waveform architectures contain many LwCCM components and only a few SCA resources.

Up to now, some memory footprint reductions have been manually validated concerning the SCA envelope and the cross connectivity. Those reductions reach around 20% on the previous examples and will be integrated in the generators.

6.2 Execution times

The execution times have been carried out with the following interaction models:

- **Local** communication on DSP with client and server components on the same thread.
- **Asynchronous** communication on DSP with client and server components on separate threads (no marshalling is performed). In this configuration, the time is measured between the client call and the beginning of the server execution and with a higher thread priority on the server.
- **Synchronous** communication on DSP with client and server components on separate threads (no marshalling is performed).
- **Emulated Remote Asynchronous** communication. The client communicates with the server on the same processing unit using the distributed middleware (marshalling / un-marshalling are included). No transport is used so that the benchmark measures the distributed protocol related processing removing the transport overhead which greatly depends on the hardware solution and the drivers (HPI, Ethernet, RapidIO, …).
- **Emulated Remote Synchronous** communication. This benchmark is performed in the same manner as the previous one with a synchronous interaction.

The IDL prototypes used for these tests are:

(a1) oneway void pushData_ow(in payload, in sType)
(a2) oneway void doIt_ow(in long, in short)
(s1) void pushData(in payload, inout sType)
(s2) short doIt(in long, inout short)

Interaction Type	Same Thread	Time
local (a1)	yes	30cycles (~50ns)
local (a2)	yes	20cycles (~33ns)
asynchronous (a1)	no	1794cycles(~2,3µs)*
asynchronous (a2)	no	1062cycles(~1,77µs)*
synchronous (s1)	no	2220cycles(~3,7µs)*
synchronous (s2)	no	2240cycles(~3,7µs)*
remote asynchronous (a1)	no	3791cycles(~6,3µs)*
remote asynchronous (a2)	no	2697cycles(~4,5µs)*
remote synchronous (s1)	no	7620cycles(~12,7µs)*
remote synchronous (s2)	no	5740cycles(~9,6µs)*

Table 3: Benchmarks

*These timings have been measured using a partition for memory allocation.

The benchmarks show that there is no overhead introduced by the LwCCM container for co-localized components in the same thread (the figures are also applicable to co-localized SCA resources on DSP because they are directly connected without using the broker solution). When the components execute in distinct threads, the timing varies depending on the allocator used to resolve the communication pattern. Using a memory partition for allocation gives better results and particularly doesn't suffer from the well-known fragmentation problem encountered with heaps that increases the time of allocations/de-allocations. The downside is the increased memory consumption compared to a heap.
It can also be noticed that some IDL parameters can increase the execution time when data copies are needed (sequence types in asynchronous requests requires a sequence construction on the server side). This mainly explains the difference between the 3rd and 4th lines in the table.
It can be noted that the code generation based on the LwCCM component container doesn't introduce overhead compared to previously hand coded solutions. The communication patterns have been captured in the framework.

7. CONCLUSIONS AND PERSPECTIVES

The dual MDE/CSBE approach using MyCCM is structuring for the waveform development. Automatic code generation offers more productivity and more flexibility to face the system variations and portability requirements. The extension of the approach on the DSP domain is an opportunity to increase the component reuse among targets but also to ease the integration step and allow fast prototyping in test environment. These flexibility and portability issues have been completely fulfilled during the Euler project.

The separation of concerns brought up by this approach allows focusing on business code development and to delegates some recurrent and error-prone issues to the code generators.

The MyCCM framework is an open framework which can be easily enriched with new functionalities or standard support and this extension capacity allow to easily capturing some design patterns which can for example allow controlling resource usage.

Some further tasks will address model analysis capabilities and particularly the use of MARTE profile [8] and also the capability to use scheduling analysis tools. Some testing issues need also to be addressed with the automatic generation of mirror test components.

8. REFERENCES

[1] SCA v2.2.2 http://sca.jpeojtrs.mil/sca.asp

[2]Lightweight CCM http://www.omg.org/spec/CORBA/3.1

[3] Active Object Pattern (Douglas C. Schmidt)

[4] OMG Deployment and Configuration http://www.omg.org/spec/DEPL/4.0/

[5] FP7 EULER project Web site http://www.euler-project.eu/

[6] Joint Tactical Radio System (JTRS) Standard Modem Hardware Abstraction Layer Application Program Interface (API). http://www.public.navy.mil/jpeojtrs/sca/Documents/SCA_APIs/API_2.13_20100629_Mhal.pdf

[7] PrismTech web site http://www.prismtech.com

[8] MARTE UML profile for modeling, analysis of real-time and embedded systems, an OMG standard: http://www.omgmarte.org/

[9] IBM® Rational® Rhapsody web site http://www-01.ibm.com/software/awdtools/rhapsody/

[10] Prismtech Spectra CX web site http://www.prismtech.com/spectra/products/spectra-cx

[11] Patent WO 2010060925; SEIGNOLE Vincent; HACHET Olivier; COUNIL Bruno; BALP Hugues(SEIGNOLE, VINCENT, ; HACHET, OLIVIER, ; COUNIL, BRUNO, ; BALP, HUGUES) ; “Method and System for encapsulating a plurality of software components compatible with the CCM standard into a software standard compatible with the SCA standard”. https://register.epo.org/espacenet/regviewer