Transparent Object Integration in Distributed H/S Systems

F. Rincón, J.Barba, F.Moya, J.Dondo, D. Villa, F.J. Villanueva, J.C. López

{Fernando.Rincon, Jesus.Barba, Francisco.Moya, David.Villa, FelixJ.Villanueva, JuanCarlos.Lopez}@uclm.es,

jdondo@inf-cr.uclm.es

Escuela Superior de Informática - Universidad de Castilla-La Mancha – Spain

http://arco.inf-cr.uclm.es/

Abstract

Based on the distributed object paradigm, we propose a design methodology where an automatically generated communication architecture provides transparent communication to any component in a SoC.

1.Introduction

Complex Systems-on-Chip can be considered as a special case of distributed system, where the application functionality is provided by a number of heterogeneous resources (hardware and software). One of the main problems in such systems is how to deal with the complexity due to the large number of components, and the different programming models for each resource. In the software domain one successful approach has been the use of communication middlewares based on the distributed objects paradigm. The middleware is an intermediate layer between the O.S. and the network that provides transparent access to objects executing in other processing elements.

Access transparency relies on the use of two special entities (the skeleton and proxy) that are automatically generated from the interface definition of the objects. The proxy takes the place of the target object in a remote invocation (between objects that belong to different processing elements). It translates the operation and parameters into a message that is routed though a communication channel. On the other side the skeleton received the messages and translates it back to a local invocation for the target object. Thus, from the object perspective remote and local invocations cannot be discerned.

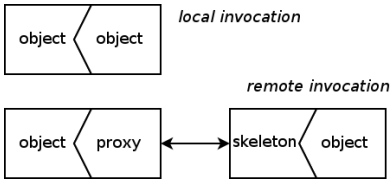

Figure 1: Local vs. Remote Invocations

The concepts described can also be extended to the hardware domain [1, 2]. Hardware components may be understood as basic objects. Proxies and skeletons can be implemented as hardware adapters between the components and the communication channel. While the system bus can be easily turned into a high performace communication engine. As a consequence any two objects in the system may communicate transparently with each other, regardless of their type of implemenation. Additionally, the system-level middleware provides a unified programming model.

2. Design Methodology

Figure 2 briefly describes the proposed design flow. First, an object-oriented model of the application must be developed. This model would be independent of any concrete implementation, and should at least specify the interfaces of the objects (class definitions), and the relationships between them. Since the model does not contain architectural information, this must be provided by a third model (deployment), where the set of resources and communication channels are specified, and a mapping between objects and the architecture is defined.

All the three inputs described, are used in a second stage to automatically generate the communication middleware. Software and hardware proxies and skeletons will be built from a template library, depending on the relationships between the objects (see section 3). Additionally, some other middleware services will also be generated in order to provide advanced communication facilites, such as off-chip transparent communication. At the same time, the information in the deployment model will be refined into a concrete architectural platform.

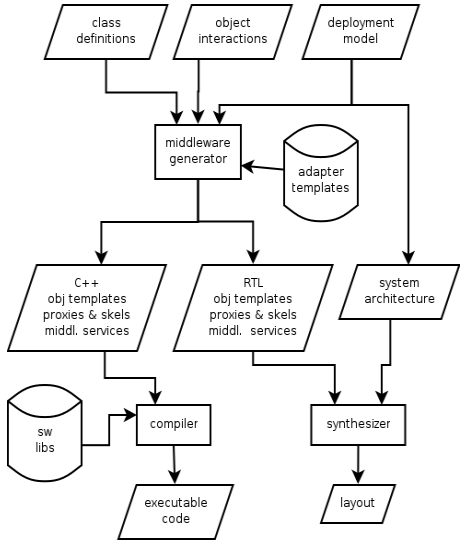

Figure 2: Design Methodology

Finally, both the system architecture and the communication middleware will be synthesized into the final hardware platform, while software objects and the software middleware part will be compiled into the executable code to

run on top of the platform.

3. A Design Example: Music Synthesizer

In the following paragraphs we present a polyphonic music synthesizer application as a case of study of the proposed methodology. Figure 4 shows the object diagram of the synthesizer (the class diagram is not show here due to space limitations). The *synth* object generates a square sound wave as a result of the composition of three different sound sources. Each source is a *voice* that plays a certain melody, described as a list of notes and note durations. Thus, voice objects are the responsible for providing the different melodies to the *synth* object. The generated sound wave (as a stream of stereo sound samples) is sent to an AC97 audio codec that generates the physical sound. A display object shows the state of the synthesizer as well as a graphical representation of the music score as it is played. Finally, the demo object is the responsible for the setup of the system.

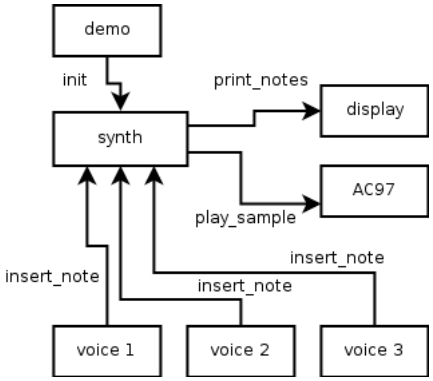

Figure 4: Design example

One of the advantages of the proposed methodology is that the architecture can be gradually refined. For example, our first approach has been a software implementation on a PC. Next we have ported it to an embedded software platform, using the Xilinx XUP prototyping board. Up to this point, all objects are local to the same processor and no special communication facilities are needed.

In the third iteration, a hardware implementation of the *synth* object has been developed. This change does not affect the rest of the objects in the application, thanks to the automatic generation of the communication middleware. Among other elements, the generated infrastructure includes a hardware skeleton and a software proxy of the *synth* object. The proxy takes the place of the original *synth* object in the software implementation, and routes transparently all invocations to the real object as a set of bus transactions. Those transactions are received by the hardware skeleton that translates them to the proper signals activations of the hardware *synth* object.

The next modification consisted in moving also one of the voice objects to hardware. Again, at the software side of the application the object is replaced by the corresponding proxy, without any consequence for the rest of the objects. On the other side, the hardware voice object uses a hardware version of the *synth* proxy, since now both are implemented in hardware. Both the hardware and software versions of the proxy generate exactly the same bus transactions for the *synth* skeleton, so for the latter point of view they are completely indistinguishable. Thus, this version of the implementation demonstrates how the same hardware object can be transparently accessed from both hardware and software clients.

A final modification consisted in moving another voice out of the embedded platform, to a remote PC. The remote voice used the software proxy of the *synth*, but this time compiled on the PC, and using a commercial software communication middleware for implementing remote invocation. Communication between the remote voice and the synth takes places through an ethernet network using the UDP protocol. Inside the board, the middleware includes a special adapter for remote communications that translates the UDP invocation into the same local bus transactions that an internal voice would generate. Finally the *synth* skeleton would activate the correspoding operations of the hardware object.

All the described situations have a hardware object as the target of the invocation. However that is not the only possibility, and all kind of interactions are supported by the middleware. It is just a matter of generating and instancing the proper proxies and skeletons. In the final implementation, for example, the display object is a software object invoked from the *synth* hardware object, and might even be outside the embedded platform.

4. Conclusion

In this work we have described how the distributed object paradigm can be applied to SoC design. The design methodology proposed relies on the automatic generation of a system-level communication middleware that provides transparent integration of hardware and software compoments. Additionally, the middleware is able to provide advanced communication capabilities, such as off-chip invocations.

For the validation of the approach, a design example has been presented, where all kind of interactions between elements have been demonstrated.

5. References

[1] F. Rincón, F. Moya, J. Barba, D. Villa, F.J. Villanueva and J.C. López, "A New Model for NoC-based Distributed Heterogeneous Systems", *Proceedings of the Parallel Computing (PARCO)*, Málaga (Spain), September 2005.

[2] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, O. Benny, D. Lyonnard, B. Lavigueur, and D. Lo, "Distributed object models for multi-processor SoC's, with application to low-power multimedia wireless systems", *Design, Automation and Test in Europe (DATE),* Munich, Germany, March 2006, pp 482-487.

Acknowledgement

This work has been funded by the Spanish Ministry of Education and Science (TIN2005-08719) and the Regional government of Castilla-La Mancha (JCCM PBI-05-0049).