Partial run-time reconfiguration based on distributed objects

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 reconfiguration facilities (location transparency, persistence and migration) are transparently provided to the applications.

1.Introduction

The distributed objects paradigm has been proposed as way to manage the complexity of complex SoC design [1]. It provides a unified programming model, and defines a set of simple abstractions that ease the integration of the large number of very different elements that are part of those systems. Such abstractions are normally implemented (through an automatic generation process) as a communication layer, usually referred as the middleware.
One of the keys of the success of communication middlewares is the ability to provide transparent communication between remote objects. This relies on the use of two abstractions called the proxy and the skeleton, that act as mediators between the objects. They take the place of the original objects (the proxy in the client side, and the skeleton on the server side) so that objects do not notice the difference between local and remote interaction. What they really do is to translate invocations into messages that are routed through a communication channel, and rebuilt as local invocations on the invoked side. Since proxies encapsulate the real location of the server object, clients never know if the server they talk to is the real object or its proxy, and neither its real location or implementation type in the latter case.
The object paradigm is specially well suited to model dynamic hardware components [2]. Two of the main problems when designing reconfigurable applications are how to guarantee system integrity and how to provide state persistence. Unlike the task-based model, the one most used, the object model provides simple solutions to both problems. One of the characteristics of objects is that the state is explicitly specified (the attributes). Any other kind of storage used by the object would correspond to intermediate computations performed by the methods, but cannot be considered as part of the state. Thus, state coherence is guaranteed by construction as far as there are no object methods in execution. It is possible then to safely stop an object in a coherent state by just waiting for the methods in execution to complete. That is not the case in the task model, where the state is not identified, and is scattered through the whole task.
Another advantage of explicitly defining the state of the object is that it is possible to automate the extraction or the insertion of the data. Hence, persistence can be provided in a systematic way and does not require any extra effort from the designer.
It is possible then to extend the transparency concept of the system-level communication middleware to provide reconfiguration transparency.

3. Reconfiguration Services

The addition of basic reconfiguration support to the system middleware requires extending the capabilities of the proxies and skeletons of dynamic hardware objects, as well as a specialized controller.
Reconfigurable objects may pass through 4 different states: IDLE, EXECUTION, STOP_REQUEST, PERSISTENCE. The object is in IDLE state by default when it has successfully been reconfigured. During this state, it is frozen and waiting for the middleware to be activated. After being acknowledged, the state changes to EXECUTION, and the object is completely functional. It will remain in this state until a stop request is signaled from the middleware, as part of the reconfiguration process. The transition from EXECUTION to IDLE happens when no method is in execution, and therefore state integrity can be guaranteed. The last state (PERSISTENCE) is used for the transmission of the internal object to or from a state memory, in order to guarantee state persistence.
This functionality is included as part of the skeleton, as an extra set of methods that complement those of the functional interface. On the other side, the proxies of reconfigurable objects also offer this methods in their interface, so that low level reconfiguration operations can be invoked by any other client object in the system.
A higher level of abstraction on the reconfiguration process is provided by a special middleware entity called the *activator*. This module is the responsible for completing the whole reconfiguration process for an object, from the initial request to the moment where the functionality has been downloaded, the state has been reloaded, and the object is activated.
On top of the activator more advanced services can be built. Hw/Sw object migration would simply require a shared persistence memory for both hardware and software implementations. High-level reconfigurable scheduling might abstract from the low-level reconfiguration tasks. Or even it should be possible to include an implicit activation service, so objects would be downloaded by the middleware as soon as an invocation to them is detected, without being explicitly reconfigured.
All this services may be provided in a transparent way to the application. From the client object point of view, there is no difference between a static (hardware or software) object

invocation and a dynamic one.

2. Design Methodology

In the following paragraphs we briefly describe very briefly 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 a specification of 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, the communication channels and the reconfigurable areas are specified. The deployment model also establishes a mapping between objects and the architecture, where some objects are marked as reconfigurable, and are initially asigned to certain reconfigurable location.
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 and the type of implementation (hardware software or dynamically reconfigurable). At the same time, the information in the deployment model will be refined into a concrete architectural platform.
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.
One of the advantages of dynamic reconfiguration is that the hardware plaftorm can evolve at run-time. It is not only possible to replace some reconfigurable objects among a set of previously design alternatives, but even new objects can be designed, implemented and downloaded after the deployment. Here the main problem is the compatibility between the predefined reconfigurable area (at compile time), with a certain number and type of inputs and outputs and the interface of the reconfigured component.
While this a non-trivial problem using an task-based approach, it is not working with distributed objects. In our case, the reconfigurable unit is not the object itself, but the object plus its skeleton and the proxies of the objects it invokes. The interface of both skeletons and proxies is just the system bus interface. Therefore, all reconfigurable areas implement the bus interface, no matter what the object interface they hold really is.

3. A Design Example: Reconfigurable Music Synthesizer

In order to demonstrate the feasibility of the approach, we have develop a polyphonic music synthesizer application (Figure 1). Initially, the *synth* object generates a 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. The generated sound wave (as a stream of stereo sound samples) is sent straight forward to an AC97 audio codec that generates the physical sound.

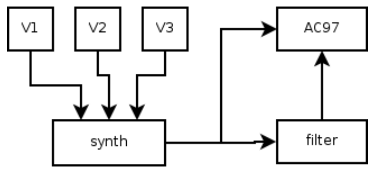

Figure 1: Design example

The *synth* object can be dynamically reconfigured, therefore the skeleton and proxy generated during the design flow include the extra set of methods for managing reconfiguration. At design time, the designer defined the state of the object (attributes) as the current volume value and the contents of three notes queues (one for each voice), so the skeleton is also able to dump automatically when requested such state to a certain memory location.
We have developed to different implementations for the syth object, one that generates a simple square sound wave and another that consumes more resources but uses a sinusoidal wave to produce a smoother sound. During the execution of the application we can then reconfigure the synth from one implementation to another. The only thing to do is to tell the generated activator to do so. Both implementations are mapped to the same memory space so voice generators are not aware of the type of which version of the synth object they are using. Furthermore, since the

4. Conclusion

In this work we have desmonstrated how the distributed object paradigm can be applied to partially dynamic reconfigurable systems. Basic reconfiguration management is provided as a service of the communication architecture and it is transparent from the application domain point of view.
A music synthesis application has been used to illustrate how the use of reconfigurability does not affect the application model. Moreover, new dynamically reconfigurable objects might be transparently added to a working system after being deployed and in run-time.

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] R. Hecht, S. Kubish, H. Michelsen, E. Zeeb and D. Timermann, "A distributed object system approach for dynamic reconfiguration". In *Reconfigurable Architectures Workshop (RAW 06)*, Rhodos, Greece, April 2006.

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).