

A new methodology takes the pain out of developing distributed heterogeneous applications.

Design Methodology Uncouples Software Architectures from Platforms

By David McCooley, Ken Hines, and Ross Ortega

As hardware manufacturers begin providing single-chip multiple-processor architectures, distributed applications are becoming increasingly common. For software developers looking to use the available processing resources efficiently, multiprocessing, and particularly heterogeneous, architectures present good optimization opportunities. Imaging, filtering, and other transformational algorithms, for example, run best on digital signal processors. Control-intensive operations and graphical user interface functionality, on the other hand, make the best use of the features of a general-purpose microprocessor. Applications that are both control-intensive and algorithmically complex, like multimedia-based products, are excellent candidates for multiple-processor architectures, such as Texas Instruments' Open Multimedia Applications Platform (OMAP).

Developing distributed software applications, however, introduces new challenges. Software architects

must consider multiple OS selections and how applications will be mapped to specific processors. In addition, specialists could be required to deal with obscure programming paradigms or very tight performance requirements. Such implementation details can invade a design to the extent that the high-level software architecture becomes blurred by the low-level partitioning decisions.

A new software methodology

called "coordination-centric" design offers a clean separation between software architecture and low-level platform issues. The methodology automatically produces efficient source code from high-level software models. It supports heterogeneous distributed architectures and allows engineers to specify low-level implementation details without reference to the software design process.

In coordination-centric design, developers focus on the creation of a robust software architecture that solves high-level issues demanded by the application, not by the hardware environment. DSP specialists, therefore, are required only to develop the algorithm(s). This approach promotes modularity and component reuse and reduces the time spent in maintaining and debugging code.

A coordination-centric software design flow starts by identifying the basic software elements—the com-

ponents and special communication “coordinators”—which are created or selected from libraries of preexisting components. Then, the components and coordinators are graphically composed to create a software design.

Once it’s been created, the design is simulated to validate its functional behavior. After that, an architectural mapping phase assigns the pieces to the computing resources, that is, components to processors and coordinators to communication mediums. Commented C code is generated that’s tuned for each operating system running on a specified processor and implements the behavior captured by the coordinators.

Let’s visit the mapping and code generation stages in more detail.

In the mapping phase, components must be assigned to processing resources—an operating system process, for example. Coordinators can be assigned to processing resources or communication mediums. Figure 1 shows three possible mappings of a simple software design to different target architectures.

The final stage of coordination-centric design, automatic code synthesis, translates the engineer’s high-level model into C code. The synthesized code provides the same functionality as the high-level design model, but important optimizations have been made. During code synthesis, component and coordinator distinctions are dissolved; components and coordinators are an artifact of logical design only. This is a

In traditional software design, a developer must select an appropriate communication mechanism, depending on the execution location of the interacting component. If the components execute in the same operating system process, shared memory is an efficient mechanism. If they’re in different processes, an operating system interprocess communication call is appropriate. If they reside on different processors,

The synthesized code provides the same functionality as the high-level design model.

significant advantage of coordination-centric code synthesis over other, more traditional object-oriented code generation approaches: by exploiting the semantic information provided by the coordination-centric model, synthesis generates code that’s much more efficient in terms of memory footprint and run-time performance.

a remote communication mechanism must be selected. All of these choices are low-level implementation decisions that complicate software architectures. In coordination-centric design, these details are dealt with before code generation and aren’t embedded in the software design.

In traditional component-based

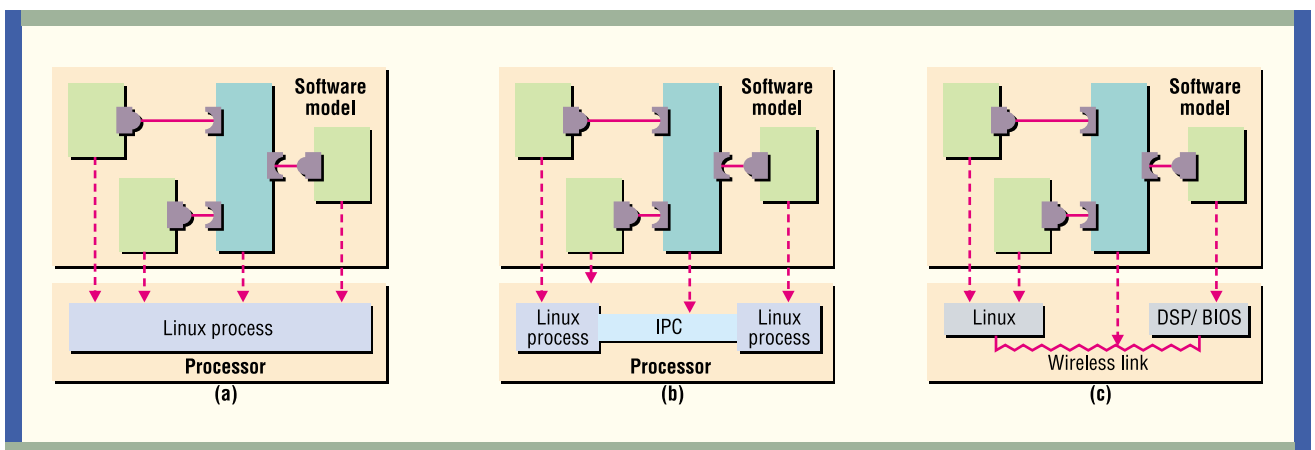


Figure 1. One of the beauties of coordination-centric design is that the software architecture remains the same with different implementations of the hardware. Software components map to processors, coordinators to either processors or communication mediums. Here, all components and coordinators map to a single Linux process (a); two components map to different Linux processes and the coordinator maps to a Linux interprocess call (b); one component is mapped to a Linux process and one to a TI DSP running DSP/BIOS, and the coordinator is mapped to a wireless communications link (c).

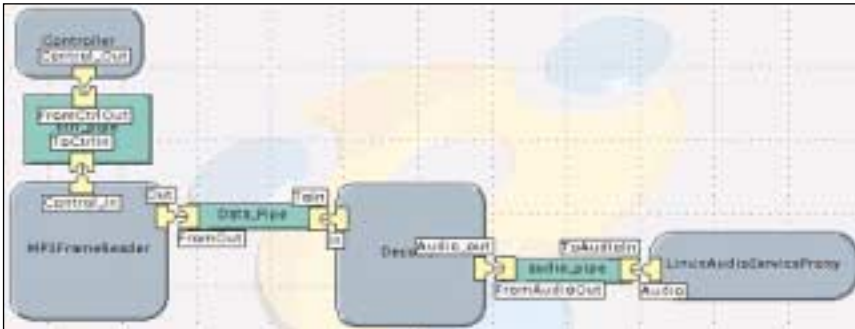


Figure 2. With Strata, you compose a software system by dragging and dropping components and coordinators and making connections through bindings. The Controller manipulates the MP3FrameReader, which in turn reads an MP3 file and provides it to the Decoder. The rounded blue blocks are reusable software components, the square green blocks the coordinators.

design methodologies, each component must contain not only its core functional behavior, but also its behavior interacting with other components. The entanglement of functional and interaction behavior forces software developers to create tightly coupled software components. If the interaction behavior among components changes, all those components must be updated to reflect the changes. These intrusive modifications are required even when the core functionality of the component isn't altered. What's more, tightly coupled software components are difficult to understand, debug, and maintain and even more difficult to reuse in different designs.

BEHIND THE BENEFITS

With a coordination-centric framework, however, software components are loosely coupled through the use of coordinators, which contain the interaction code between components. A design rule requires a coordinator between any interacting components.

Thus a coordination-centric framework encourages reuse by allowing system components to be interchanged in a "plug and play" fashion; indeed, entire subsystems can

be designed independently from other system components. Also, the debugging phase is greatly reduced because the correctness of the software functionality is separated from its interaction behavior. Essentially, both debugging and maintaining software are simplified because there are no "back door" interaction paths with the components. All communication and interaction is

explicit and clearly defined.

A distributed MP3 player application is a good example of how coordination-centric design works. Let's look at an MP3 player we designed using Strata, a commercial implementation of coordination-centric design.

MP3 PLAYER

The design steps were incorporating existing code into components, using models to validate the behavior of the architecture, mapping the design to a single processor and synthesizing the code. The very last steps were to map the same design to a distributed-processor architecture and synthesize the code for each processor.

We used a PC running Linux as the development host for Strata, for design entry and behavioral simulation of the system prior to mapping. The same PC served as the target and to create a Linux executable that plays MP3 songs from a file system on the PC. For the distributed-processor mapping, we used an

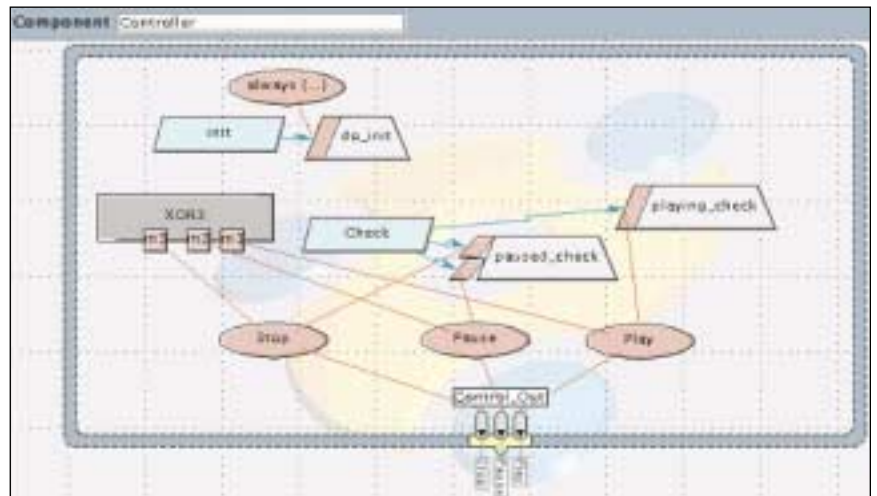


Figure 3. An action triple model allows software contained in action trapezoids to be executed upon the occurrence of certain trigger events, provided that the related mode of operation is true. For instance, within the Controller, when the Play mode is active and the Check event occurs, the code snippet inside playing_check is executed.

Ethernet crossover cable that connected the PC to a Texas Instruments TMS320C6711 DSP board. The distributed mapping of the player placed the control and file reader portion on the PC and the decoder and playback on the DSP board. We loaded the synthesized code for the DSP board using TI's Code Composer Studio IDE.

The first task was to partition the player functionality into coordination-centric components and coordinators. Because of the streaming nature of the application, the coordinators required were simple data pipes. (Figure 2 shows a Strata tool view of the software design for the MP3 player.) To demonstrate the main decoding algorithm, we wrote a simple C-based control GUI to start, pause, and stop the song.

Pushing into the Controller component, shown in Figure 3, highlights some interesting features of the coordination-centric design framework. The graphical syntax captures the event-driven programming model, which uses modes, events, and actions. Modes are internal state variables that guard the action. Actions can turn modes on or off, generate events, modify variables, and make foreign subroutine calls. Events are generated either automatically by the system or explicitly by an action. The combination of a mode, an event, and an action constitutes an "action triple." For example, when the Play mode is true and the Check event arrives, the action `playing_check` is triggered.

USING LEGACY CODE

A foreign subroutine is one written in C or Java. A major advantage of coordination-centric design is that such legacy code can be intermingled with new software components. The `playing_check` action code, below, communicates with the GUI controller via the `MP3_Control` foreign

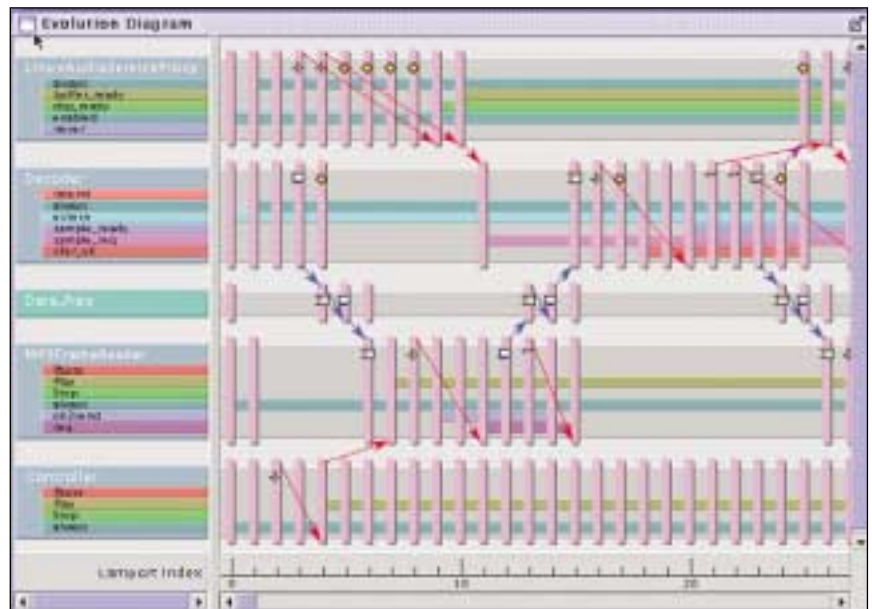


Figure 4. Strata provides a visual tool, the evolution diagram, that lets you follow the state, control flow, and data flow of a software architecture, allowing time-saving debugging long before the mapping to actual hardware. Here the Decoder and MP3FrameReader exchange messages (blue arrows) via the Data_Pipe. At index 2, an event in the Controller requests that the Play mode become active (red arrow). The Play mode is activated at index 4, and this information is transmitted to the MP3FrameReader, whose Play mode also is activated.

call, which returns the identity of the button pressed. After activating the corresponding mode, the action generates the Check event.

```
foreign function int Mp3_Control ();
int result;
result = Mp3_Control ();
if ((result == 1)) {
    +Play;
} else {
    if ((result == 2)) {
        +Pause;
    } else {
        if ((result == 3)) {
            +Stop;
        }
    }
}
->Check ();
```

The rectangular box named XOR3

in Figure 3 is a user-defined constraint block. It enforces a mutual exclusion constraint between the modes Play, Pause, and Stop, which are connected to the inputs m1, m2, and m3, respectively. Constraints clearly declare and enforce the intended use of a component. In standard programs, in contrast, constraints are usually embedded into the structure of the code or contained in a comment to remind the developer and others about the intended use of a body of code. Also, note that like components and coordinators, constraints are reusable elements of a software design.

Coordination-centric design provides a very powerful abstraction in the ability to share states among components without committing to a communication mechanism. Com-

ponent and coordinator interactions occur via interfaces. Figure 3 shows the Controller's graphical connection to its Control_Out interface. Graphically the modes Play, Pause, and Stop are bound to this interface and therefore shared with the MP3Frame_Reader via the coordinator ctrl_pipe, as shown in Figure 2.

INDEPENDENT SIMULATION

To validate the functional behavior of the design, we first simulated it independent of operating system and hardware issues. By debugging the design at this level of abstraction, we gained confidence in the algorithm's correctness. In a traditional debugging environment, an error could be in the algorithm; it could be an

improper use of the operating system or a communications protocol; it could be a hardware bug or a timing problem; and so on.

The developer must consider all the possibilities simultaneously. To ease the task, the coordination-centric framework provides a visualization called "evolution diagrams" that graphically illustrate interactions among the components and coordinators, as shown in Figure 4. (Note that this simulation isn't a real-time execution of the system.) Each horizontal trace is a component or coordinator. Within a component or coordinator, colored horizontal bars represent the component's or coordinator's mode. Vertical bars are events. Colored arrows

among the traces show control communications, message sends, and variable propagation. Of particular interest are the control and variable arrows, which show interaction behavior that wasn't hand-coded by the developer but instead generated automatically.

The sound quality demonstrated by the model simulation proved that the architecture was functioning properly.

TARGET MAPPING

Confident in the correctness of the architecture, we created a Linux MP3 player by mapping the design to a single Linux process. Since coordinators are logical constructs, the ones in Figure 2 were optimized



Four of a kind!

Being first to market is a sure bet when your hand includes cards like the Quatro6x...

Features

- ▶ Four 160 MHz TMS320C6701 DSPs (floating-point)
- ▶ Plug-n-Play PCI Interface
- ▶ FIFO link interconnects between processors
- ▶ Three FIFOPort interfaces
- ▶ Comprehensive C/C++ cross development tools & algorithm templates
- ▶ Windows 9x/NT/2000 drivers

Applications

- ▶ Numeric coprocessing
- ▶ Video coprocessing
- ▶ Large-scale data reduction
- ▶ Huge, multiprocessor arrays

Get your data sheets now!
Visit the Quatro6x site at
www.innovative-dsp.com/q6x

805.520.3300 phone
www.innovative-dsp.com

Innovative Integration
... real time solutions!

Quatro6x

away during code generation, in this case because all interaction occurs in the shared address space of a Linux process. Code generation created commented C code and a corresponding make file. The code was then compiled with gcc to create the binary image. Because the simulation foreign subroutines were written for Linux, we reused them for the target. The resulting MP3 player executable played in real time.

As described earlier, we then distributed the design across a Linux PC and a C6711 DSP evaluation board connected by an Ethernet crossover cable. The distributed MP3 player plays MP3 songs by retrieving them from the Linux side, transmitting them to the DSP board, decoding them, and playing them through an external speaker.

Figure 5 shows the mapping table for the distributed player. We mapped the decoder component to the DSP board, the connecting coordinator to the TCP link, and the remaining control and file aspects to the Linux computer. Code generation created two sets of C files, one for the Linux side and one for the DSP/BIOS side, as well as TCP calls for Linux and for DSP/BIOS. Using Code Composer Studio, we compiled the C files for the DSP board and downloaded the binary image onto the board. We compiled the Linux side with gcc, then ran both executables and listened to an MP3-encoded song.

RESULTS

The memory requirements for the distributed MP3 player synthesized software are as follows: for the Linux MP3 player, the Linux footprint was 233 kilobytes; for the distributed MP3 player, it was 119 KB, and the DSP footprint was 805 KB.

The large DSP footprint needs further explanation. Because we had essentially a "raw" board, the entire

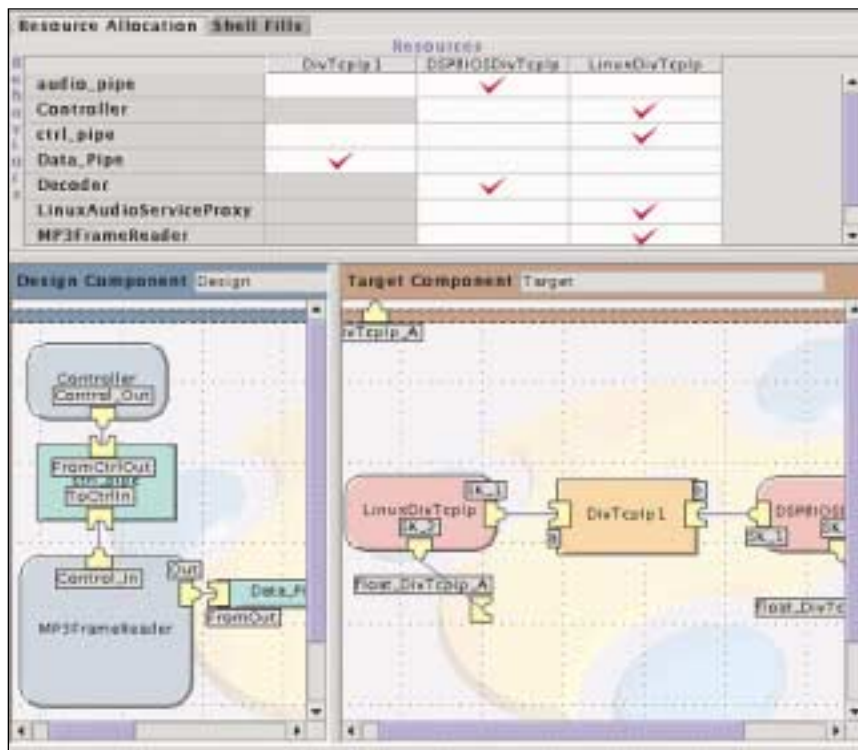


Figure 5. A software architecture (bottom left) is mapped to a hardware architecture (bottom right) via a mapping table (top). To repartition the behavioral elements, you simply check the box associated with the computational resource available on the target. Here, the Decoder and audio_pipe are mapped to the DSP, the Data_Pipe is mapped to a TCP link, and the rest of the components are mapped to Linux.

software image had to be downloaded. The TCP/IP stack provided with the development kit—downloaded to drive the Ethernet connection—is 301 KB (37% of the DSP image). Other DSP/BIOS libraries and objects required an additional 68 KB (8%). The DSP architecture requires the alignment of various data structures; it enforces the alignments by introducing “holes,” or gaps, in memory, which took up 285 KB (35%). We had to write initialization code to bring up the TCP/IP stack, which required 20 KB (2%). The legacy code, including the MP3 decoder algorithm itself, required 96 KB (12%), and the generated synthesized code 35 KB (4%). ◆

David McCooley (david.mccooley@consyant.com) is a senior software engineer at Consyant Design Technologies, Inc., focusing on platform differences and how to capture them in APIs. For the previous 12 years, he worked on the internals of the LynxOS RTOS at Lynx (later to become LynxWorks). Before that, he was a member of the technical staff at AT&T Bell Labs, working on the Access Network System (ANS). Ken Hines (ken.hines@consyant.com) is Consyant's chief scientist, a vice president, and co-chairman of the board; Ross Ortega (ross.ortega@consyant.com) is chief technology officer, a vice president, and co-chairman of the board. Hines and Ortega founded Consyant based on their doctoral research.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265