Programming Specifications in CAL

Edward D. Willink[1], Ed.Willink@uk.thalesgroup.com

Thales Research and Technology (UK) Limited, Reading, England

Johan Eker, johane@control.lth.se

Lund Institute of Technology, Lund University, Sweden

Jőrn W. Janneck, janneck@eecs.berkeley.edu

University of California at Berkeley, CA, USA

1 Introduction

Object-oriented technology (using UML) has indeed given us a better handle on complexity than previous technologies. It achieves this by strengthening data encapsulation and providing some functional encapsulation. However full encapsulation of behaviour is missing, and this is what more component-oriented approaches seek to address.

Components have been used informally for systems engineering as the blocks in block diagrams, since long before computers, let alone UML, ever existed. Indeed computers are generally designed using block diagrams. A block diagram, such as that shown in Figure 1, comprises boxes representing functional elements and arcs representing connecting communication paths. In contrast to UML diagrams, arcs connect at named ports identifying specific interfaces and invocation mechanisms.

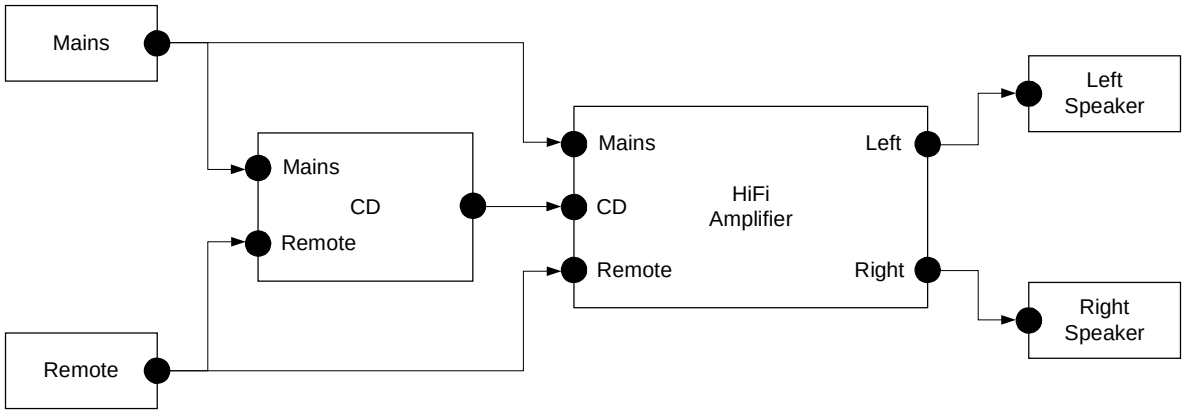

Figure 1 Block Diagram

The diagram shows a HiFi system. This is an overtly hardware example for ease of comprehension. Of course, in software systems, the blocks will be software components for which we will seek to create the illusion of independent entities by time-multiplexing the resources of a comparatively small number of processing elements.

Block diagrams are widely used for Embedded Software [5] and more especially for Digital Signal Processing, where the relationship between alternate hardware and software realisations is clear. The block diagrams, even with informal semantics, are useful for rapidly conveying the structure of a design and partitioning the functionality into sub-systems or components. However formal semantics are necessary to avoid misunderstandings and essential for automated code generation. We must define what, when and why information flows along each communication path. It is only when the very diverse natures of all these properties can be encapsulated within a hierarchical component, that we can use hierarchy as a tool for composition. The diagram semantics

[1] Funded by the Thales Signal Processing Environments and ARchitectures Program.

must be able to describe the scheduling and concurrency of the fully parallel system depicted in a block diagram, so that we can reliably create the illusion of concurrency. The fully parallel system, depicted in a block diagram, therefore represents a specification of what must be done by a typically much less parallel implementation.

The Model Driven Architecture (MDA) recognises the need to refine a specification model into an implementation model, but as yet UML fails to recognise the major advances in component modelling that have been possible in the more restrictive field of signal processing. The advances towards platform independent models are hampered by the lack of support for scheduling independence. Direct usage of operating system concepts, such as tasks or mutexes, imposes a particular implementation and consequently excludes alternative implementations. An inclusive specification of what must be done is required.

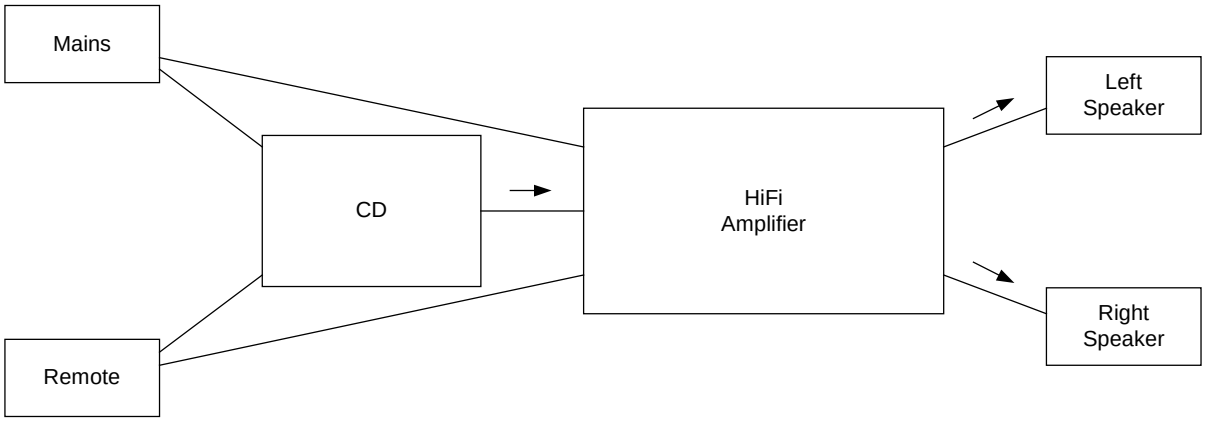

Figure 2 UML collaboration diagram

The corresponding UML collaboration diagram, shown in Figure 2, denotes the relationship between the amplifier and the speakers, but fails to distinguish the very distinct interfaces used by the speakers, mains supply or CD player. It is necessary to provide informal accompanying text to explain the way in which information flow along the CD to amplifier link interacts with information flow along the speaker links. This text is informal, external to the diagram; it acts as a severe impediment to hierarchical decomposition of the diagram into fully encapsulated components.

The block diagram incorporates the extra detail necessary to describe the context of a component. The rules for communication between the blocks are determined by a Model of Computation (MoC) [7] that applies to all interactions within a diagram. Different models are appropriate for different applications. The commonest models in software systems are Data Flow (DF) for compute intensive operation, or Discrete Event (DE) for message intensive operations. It is often appropriate to use a mixture of models in different parts of the system, thus a DE model may be appropriate for handling the sporadic commands coming from the remote control zapper, while a DF model is appropriate for the incessant signal processing.

Tool support for block diagrams is well established and current research concentrates on the support for a mixture of models of computation, in particular the arbitrary hierarchical intermixing of state machines and block diagrams [2]. These approaches were brought together in the proposal for a Waveform Description Language [9].

Generation of code from a block diagram involves synthesis of an appropriate scheduling and communication framework around the pre-existing code for each block. In tools such as Ptolemy Classic, code generation operates using a library of code templates for each block, with different code templates for each MoC and in some cases for distinct data types as well. The existence of many code templates, for nominally the same purpose, creates consistency and maintenance difficulties. In Ptolemy II [3], where each block is referred to as an actor, there is much more polymorphism; a single code template may support a variety of MoCs and data types. However the extreme polymorphism of code

templates written in Java imposes severe challenges to generation of efficient production quality code.

2 Caltrop

The Caltrop project [1] is concerned with preserving the flexibility of the Ptolemy II model while defining the behaviour of each actor in ways that allow the external interactions to be analysed., This provides the insight necessary for synthesis of an efficient code framework. As will be seen, the use of XSLT also supports a very open architecture allowing for component optimisation, folding and efficient target code generation.

Block diagrams are hierarchical and the user of a block should not need to know whether a block is composed hierarchically out of many blocks or is directly implemented as a leaf actor, which CAL defines. If CAL is expressive enough to define actors, it must be possible to re-express a composite hierarchical actor as a leaf actor in CAL. Likewise, if XSLT is powerful enough to perform all CAL transformations, it must be possible to use XSLT to combine a number of smaller CAL definitions to create a larger definition, and ultimately after sufficient combination to compose the CAL definition of the entire system. This system definition can be converted to efficient code for the target processor. We will demonstrate these points.

2.1 Language

CAL is an actor definition language, whose proper exposition would take too much space, so only very simple examples will be used with fairly intuitive syntax. A two input adder can be defined as:

```
actor Add[T] () T in1, T in2 ==> T out1
:
    action in1:[a], in2:[b] ==> out1:[a + b] end
end
```

The **actor** line at the top of the actor construct, defines the port signature of the Add actor with a generic type parameter T and no value parameters. There are two input ports in1 and in2 each of type T and an output port out1 also of type T. This line therefore defines the connectivity and parameterisability of the component.

Each **action** construct, just one line in the example, defines a possibly partial pattern of input values and a consequent response at some outputs whenever the input pattern appears. In the example, the input pattern comprises a single value at each input. The values are bound to the labels a and b within the action so that the expression a+b at the output is sufficient to specify that the output response is the sum of the inputs. The action therefore has a very direct association with the firing rules of Kahn Process Networks [6].

More complicated actors may involve state, functions, procedural code and multiple actions.

CAL avoids a bias towards any particular MoC, by leaving all MoC specific behaviour external to the actor definition. The external MoC affects how the actor is invoked, not how the actor responds once invoked.

CAL is a source representation and so compositions using XSLT are source to source program transformations,

2.2 Composition

In order to create the composite CAL for a composition of actors, it is necessary to combine the CAL code for each actor, and for the composing MoC. The way in which this can be done will be shown by using a simple example. We will synthesize code to evaluate sqrt(x * x + y * y) by combining two instances of a squarer:

```
actor Sqr[T] () T in1 ==> T out1
:
```

```
        action in1 : [a] ==> out1 : [a * a] end
    end
```

one instance of the adder above, and one instance of a square-root actor, which acts as a wrapper for the square root function::

```
actor Sqrt[T] () T in1 ==> T out1
:
    action in1:[a] ==> out1:[sqrt(a)] end
end
```

A graphical representation of the required composition is shown in Figure 3.

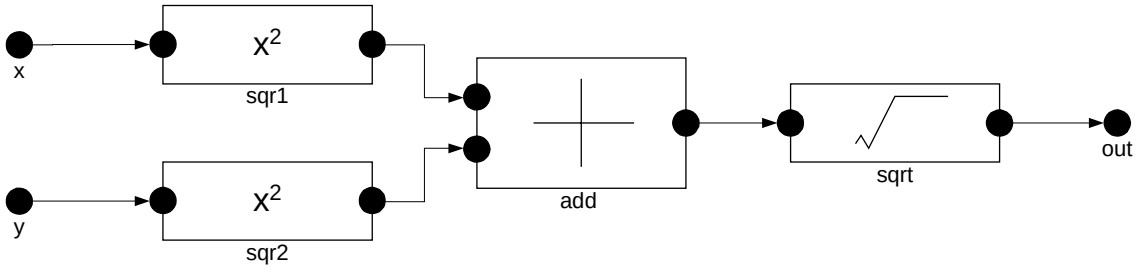

Figure 3 Complex Absolute Composite Actor

A MoML[2] net-list is extracted from the graphical representation to drive the composition and produce the composed CAL source:

```
actor cabs [T] () T x, T y ==> T out
:
    action x : [r1], y : [r2] ==> out : [r6]
    var
        T r3;
        T r4;
        T r5;
        T r6;
    do
        r4 := r2 * r2;          // sqr2
        r3 := r1 * r1;          // sqr1
        r5 := r3 + r4;          // add
        r6 := sqrt(r5);         // sqrt
    end
end
```

This composed code uses more elaborate syntax for the body of the composite action; the result is now determined by the fourth of four local variables `r3` to `r6`. The trivial calculations for each original actor are performed one at a time, to successive local variables. It is easy to see that this can be transformed to C or VHDL code without too much difficulty.

Combining the actor code just requires a little careful renaming to avoid erroneous name capture. Incorporating the MoC requires code to be synthesised to enforce the MoC.

For the case of Synchronous Data Flow it would be possible to impose an MoC relatively easily by associating temporary variables with each connection and imposing a left to right evaluation order. In order to handle a wider range of MoCs we pursue a more generalised approach.

The data flow paradigm requires the insertion of unbounded FIFOs in each connection, and these may be modelled in CAL by an infinite list with associated get and put actions. The FIFOs on the external connections can be eliminated, since the instantiating MoC describes a data flow composite and so the invoking context will be responsible for provision of any input/output buffering. Analysis of the token consumption and production rates for the actors reveals that the internal FIFOs are also redundant and can be removed. Analysis of the causal constraints and imposition of a single processor scheduling constraint is then sufficient to produce an obvious schedule, enabling the

[2] The XML dialect used by Ptolemy II.

transformations to produce the composite leaf actor code.

The composite is only valid for invocation under the composing MoC. In this respect the composition is inferior to a fully fledged CAL actor, which is valid for invocation under any MoC. However this restriction is not significant, since the restriction is incurred after the required MoC is known.

The formation of the composite is essentially free of instantiation overheads. It is just a little profligate with temporary variables, but optimisation of temporary variables, particularly those used in a disciplined Static Single Assignment fashion, is within the capabilities of register colouring optimisations.

The use of source to source transformations has therefore allowed high level semantic optimisations to be performed at the source level, leaving low level optimisations to be performed by the back end compiler.

2.3 Transformation Infrastructure

CAL has a textual language presentation for ease of editing, and an equivalent XML dialect (CalML). CAL compilation starts with a translation to CalML and then proceeds using a series of XSL transformations that steadily

- perform semantic validation
- eliminate the more sugary syntaxes
- normalise
- perform type inference, constant folding, ...
- generate code in a target language

With the entire compiler operating on a single XML schema and all transformations accessible as XSLT, the architecture is open for addition of user-defined transformations.

The amount of optimisation that can be performed on individual actors is limited. However once a CAL actor is connected for use in a particular application, it is possible to apply source to source transformations to specialise the generic actor to create an application specific actor for the required application. Unused outputs may be stripped out. Unconnected or constant inputs may be exploited to simplify the behaviour. Type inference may be used to simplify code, or synthesise loops when a generic type parameter turns out to be an array rather than a scalar. Analysis of the scheduling context may identify concurrent or non-concurrent connections and consequently eliminate actions that cannot fire under the instantiating MoC.

These are all source to source transformations that are relatively easily performed by XSL transformations at the source level where the meaning is clear. It is harder to perform this kind of optimisation once the code has been translated to a target language whose compiler may not understand the higher level scheduling concepts and redundancies.

XSLT is not the most concise of programming languages, so we have developed a front end translator from a more user friendly NiceXSL [8] format.

2.4 Constructive scheduling

In conventional implementation approaches using classes, scheduling integrity is imposed by introduction of protective mechanisms such as mutexes. These are one-sided run-time attempts to ensure correct behaviour, and while they protect the data, they may easily contribute to deadlocks.

The composition approach using CAL synthesises the required mutexes automatically. The language definition imposes a prohibition on concurrent access of shared state by two actions. This prohibition must be observed by the subsequent schedule, and may result in a mutex when necessary, but may result in no code at all when a schedule can be constructed that is inherently exclusive.

The two ends of a communication link are normally programmed independently, again providing considerable scope for deadlocks. Using the CAL approach of constructing a schedule to satisfy a communication specification, the problems of deadlock are much reduced. Further research is required to see to whether the more restrictive context can totally eliminate them.

These benefits arise from synthesising code to meet a specification.

3 Summary

Component-oriented design approaches are in widespread use for signal processing, and its block diagram languages are given solid semantics by Models of Computation. Limitations in the way in which the behaviour of blocks are defined motivated the CAL actor definition language. Consideration of the way in which well defined actors are composed, shows that the MoC is the composition function, and that consequently the scheduling mechanisms required by the MoC can be synthesised automatically with actor definition and MoC serving as the program specification.

The MoC defines both the block diagram semantics and the code generation transformations. With subtly different transformations applicable to different target environments, meta-modelling of the MoC would be beneficial [3].

The time has come to stop programming better implementations and to start to program specifications.

CAL and more especially its underlying XSL technology provides for greater openness, which paves the way for smart implementation approaches to be defined as re-usable transformations, rather than being realised as a manual knitting that inhibits re-use.

4 References

[1] J.Eker and J.W.Janneck, “Introduction to the Caltrop actor language”, http://www.gigascale.org/caltrop/docs/CaltropWhitePaper.pdf

[2] A.Girault, B.Lee, and E.A.Lee, ``Hierarchical Finite State Machines with Multiple Concurrency Models,', IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems, Vol. 18, No. 6, June 1999 (revised from Memorandum UCB/ERL M97/57, Electronics Research Laboratory, University of California, Berkeley, CA 94720, August 1997). http://ptolemy.eecs.berkeley.edu/publications/papers/99/starcharts

[3] A.Ledeczi, A.Bakay, M.Maroti, P.Volgyesi, G.Nordstrom, J.Sprinkle , and G.Karsai, “Composing Domain-Specific Design Environments”, Computer, 44-51, November, 2001. http://ransom.vuse.vanderbilt.edu/academic/publications/Composing Domain-Specific Design Environments.pdf

[4] E.A.Lee, "Overview of the Ptolemy Project", Technical Memorandum UCB/ERL M01/11, University of California, Berkeley, March 6, 2001. http://ptolemy.eecs.berkeley.edu/publications/papers/01/overview

[5] E.A.Lee, “Embedded Software”, In M. Zelkowitz (ed), Advances in Computers, Vol. 56, Academic Press, London, 2002. http://ptolemy.eecs.berkeley.edu/publications/papers/02/embsoft/embsoftwre.pdf

[6] E.A.Lee and A.Sangiovanni-Vincentelli, ``A Denotational Framework for Comparing Models of Computation'', ERL Memorandum UCB/ERL M97/11, University of California, Berkeley, CA 94720, January 30, 1997. http://ptolemy.eecs.berkeley.edu/publications/papers/97/denotational

[7] E A.Lee and Y.Xiong, "System-Level Types for Component-Based Design", First Workshop on Embedded Software, EMSOFT2001, Lake Tahoe, CA, USA, Oct. 8-10, 2001. http://ptolemy.eecs.berkeley.edu/publications/papers/01/systemLevelType

[8] E.D.Willink, “NiceXSL”, September 2002, http://www.gigascale.org/caltrop/NiceXSL

[9] E.D.Willink, “The Waveform Description Language”, In W.Tuttlebee (ed), Software Defined Radio: Enabling Technologies, John Wiley, 2002. http://www.computing.surrey.ac.uk/personal/pg/E.Willink/wdl/documents/MobileVceChapter.pdf