Verification or Validation of Hardware-Software Codesigns?

Richard G. Nicholls and John Ainscough
Department of Electrical and Electronic Engineering
Manchester Metropolitan University
Chester Street, Manchester. M1 5GD

Abstract

Hardware-software codesign is particularly well suited for real-time embedded systems. It is therefore imperative that the design implements the specification correctly. In this paper the method of validation or verifying designs is discussed, and tools are presented that can be used to provide a framework for verifying codesigned systems. Also discussed is how these tools can be integrated for such a framework.

Introduction

Codesign is used to develop systems which result in a final implementation consisting of both hardware and software components. Codesign attempts to produce a final implementation that has an optimum partition between hardware and software within specified constraints. As it is not usually possible to explore all possible partitions other methods are used to reduce the number of partitions evaluated. Most approaches to codesign rely on a method of synthesis, simulation and re-evaluation of the partition, based on the simulation results. It is believed that if the number of iterations to reaching the optimum partition is to be reduced, then a method is required that produces a good initial partition. However, as stated before, re-partitioning relies on simulation techniques and hence the initial partition is usually designer oriented.

Simulation techniques are, by themselves, inadequate for verifying a system. With a suitable data set, simulation may be used to validate that the system performs correctly, but only for those possible stimuli. Dijkstra[1] remarked that "Testing only establishes the presents of faults not their absence". It would be more appropriate then if for a given implementation it could be verified as meeting a given specification, which contains all known constraints. Formal proof techniques can be used to achieve this, and were pioneered by people such as Gordon[2], Hunt[3] and Boyer and Moore[4]. Formal methods allow a specification to represent an unambiguous model of a system and to be free of design detail. By using rules of logic, it can be proved formally that an implementation matches its specification. Verification is therefore potentially a mechanical process. The HOL theorem prover is a system providing an environment for applying formal proof techniques using higher-order logic. The HOL system is discussed later.

In order to integrate formal methods into the design process it is necessary to provide a suitable framework that supports a high-level of behavioural abstraction. The Ptolemy system provides a unified base for the designer, using small interconnecting graphical blocks. An overview of the Ptolemy system is presented later.

This paper goes on to describe how Ptolemy can be extended and incorporate the HOL theorem proving system. Hence providing a framework for verified codesign.

Ptolemy

Ptolemy[5] is a software environment that supports heterogeneous system specification, simulation and design. It is ideal for hardware-software codesign as heterogeneity is a key characteristic of these systems. Ptolemy makes use of object oriented techniques to achieve the following goals:

- *Agility*: Support distinct computational models, allowing each subsystem to be simulated and prototyped in a manner that is appropriate and natural.
- *Heterogeneity*: Allow seamless coexistence of distinct computational models for the purpose of studying the interactions between their subsystems
- *Extensibility*: Support seamless integration of new computational models and allow them to interoperate with existing models without making changes to Ptolemy or existing models
- *Friendliness*: Provision of a modern graphical interface using a hierarchical block diagram style of representation

It is the first three of these that are of interest. Ptolemy uses a non-dogmatic kernel that does not impose any particular computational model. However, by using these object oriented techniques it provides a framework for co-ordinating different computational models.

The basic unit of modularity in Ptolemy is the block, these communicate via portholes. A block contains a go() method which typically examines data present at its input portholes and generates data on its output portholes. A scheduler directs the invocation of the block, determined by their operational semantics within a network. Blocks communicate using streams of particles, which form the base type for all messages.

Ptolemy uses hierarchical methods to manage complexity in large systems. At the lowest level objects in Ptolemy are of type Star, derived from Block. A Galaxy, also derived from Block, contains other blocks such as other Galaxies or Stars. A Universe contains a complete Ptolemy application and is of type Galaxy.

The basic mechanism in supporting heterogeneity in Ptolemy is the extension of the hierarchy of Stars, Galaxies and Universes to include objects called Wormholes. From the outside the wormhole look monolithic, like a Star, but inside it looks more like a Galaxy in a different domain. The scheduler on the outside treats as a Star, but internally it contains its own scheduler. These schedulers need not abide by the same model of computation.

In order for particles to pass from one domain to another (into or out of a wormhole) they do so through an Event Horizon. The event horizon manages any format translations that may be necessary to bring together two models of computation.

Targets, again derived from blocks, controls the execution of an application whether it is a simulation or synthesis process. In a simulation-oriented application it would typically invoke a scheduler to manage star invocation. Whereas in a synthesis-oriented application it can generate and compile/assemble code, and possibly download and manage the execution.

A Domain consists of a set of stars, schedulers and targets that conform to a common computation model - the operational semantics that govern how blocks interact. Subdomains can exist that implement a more specialised model of computation than the outer domain. Examples of domains are:

- DDF - Dynamic Data Flow
- SDF - Synchronous Data Flow
- BDF - Boolean Data Flow
- DE - Discrete Event

The Ptolemy system is fundamentally extensible. Users can extend domains by writing their own stars or can themselves develop complete new domains for a computation model.

The HOL System

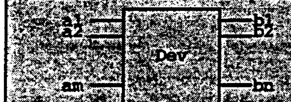
The HOL system[6] provides a powerful environment for interactive theorem proving in higher-order logic. Higher-order logic is an extension of predicate logic that allows quantified variables to range over functions and predicates. This allows other formalisms to be represented within it via their semantics.

The primary interface to HOL is the functional programming language ML (Meta Language). The three ML types that form the interface to the logic are: type, term and thm. Values of these types are data-structures that represent types, terms and theorems of the higher-order logic in ML. There are five predefined ML identifiers of type thm which provide five basic axioms. The only way to generate more theorems is to apply ML functions. By doing this the theorem prover can be programmed and extended. Theorem proving tools are functions in ML. In the core of the system there are only eight such functions which perform the eight rules of inference that make up the deductive system of higher-order logic.

When the HOL system is built hundreds of theorems are preproved and stored in theories, including booleans, individuals, numbers, sums, lists and tress. Theories consist of types, constants, definitions, axioms and an explicit list of theorems that have been proved from the axioms and definitions. Typically an interaction with HOL will result in a new theory, usual an extension or combination of existing theories.

Representing behaviour with predicates

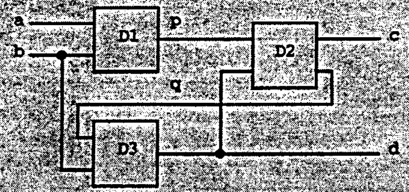
A device is a 'black box' with a specified behaviour, for example:



This device is called Dev and has external lines a1,a2,...,am,b1,b2,...,bn. These lines correspond to the 'pins' of an integrated circuit. When the device is in operation each line has a value drawn from some set of possible values. Different kinds of device are modelled with different sets of values. The behaviour of device Dev is specified by defining a predicate Dev (with $m+n$ arguments) such that Dev($a1,a2,...,am,b1,b2,...,bn$) holds if and only if $a1,a2,...,am,b1,b2,...,bn$ are allowable values on the corresponding lines of Dev.

Representing circuit structure with predicates

Consider the following structure (called D):



This device is built by connecting together three component devices D1, D2, and D3. The external lines of D are a, b, c and d. The lines p and q are internal and are not connected to the 'outside world'. (External lines might correspond to the pins of an integrated circuit, and internal lines to tracks.)

Suppose the behaviours of D1, D2 and D3 are specified by predicates D_1, D_2 and D_3 respectively. How can we derive the behaviour of the system D shown above? Each device constrains the values on its lines. If a, b and p denote the values on the lines a, b and p, then D1 constrains these values so that $D_1(a,b,p)$ holds. To get the constraint imposed by the whole device D we just conjoin (i.e. $\wedge$ together) the constraints imposed by D1, D2, and D3; the combined constraint is thus:

$$D_1(a,b,p) \wedge D_2(p,d,c) \wedge D_3(q,b,d)$$

This expression constrains the values on both the external lines a, b, c and d and the internal lines p and q. If we regard D as a 'black box' with the internal lines invisible, then we are really only interested in what constraints are imposed on its external lines. The variables a, b, c and d will denote possible values at the external lines a, b, c and d if and only if the conjunction above holds *for some* values p and q. We can therefore define a predicate D representing the behaviour of D by:

$$D(a,b,c,d) \equiv \exists p\ q.\ D_1(a,b,p) \wedge D_2(p,d,c) \wedge D_3(q,b,d)$$

Thus we see that the behaviour corresponding to a circuit is got by:
- Conjoining the constraints corresponding to the components, and
- existentially quantifying the variables corresponding to the internal lines.

Panel A Extract from Gordon[2]

An important part of the HOL system is the large collection of tactics as this supports goal directed proof. These tactics rewrite the goal according to some preproved theorem or definition, remove unnecessary universally quantified variables from the goal, and split equalities into two implicative subgoals.

The HOL system can be used to verify both software and hardware. A number of microprocessors have been verified[7,8], some not completely, with it. Current work of interest to us is looking at embedding hardware description languages in HOL[9], and Logical Formalisation of Hardware Design Diagrams[10].

With the possibility of extending this to verify systems built from these blocks by their conjunction.

Creating a Framework for Verified Codesign

It has been shown in previous work[11] that codesign is most effective at the functional block level. It is with this in mind that the verification of a number of small functional blocks from specification to underlying designs in both hardware and software is been undertaking. The HOL system is being used to formally prove these designs meet the specification for each functional block. Hence creating theories for the software and hardware verification for each functional block. Having proved the functional block it then needs to be represented within Ptolemy. When these functional blocks are connected within an application in Ptolemy, the verification of the whole system design would then be a conjunction of the smaller verified functional blocks, connectivity information being provided by Ptolemy. Similar work to this for hardware circuits is demonstrated in Panel A.

How the Ptolemy environment can be extended to implement the above method is being investigated. The obvious solution appears to create a new Ptolemy domain, with it's own scheduler, target and stars, based on the HOL theorem proving system. The functional blocks represented within this domain may be associated with similar blocks from simulation domains within Ptolemy. The initial implementation of this domain would allow the designer to choose the designated representation for a particular functional block, probably based on the results of a simulation. Whether the functional block has a selectable implementation or whether they are chosen from two subdomains, one for hardware the other for software, is still under investigation. Communication between the hardware and software blocks has to be investigated to see how it will affect the verification process.

The role of the scheduler within the new domain is still being considered. It could be used to create the necessary conjoined definition of the design for verification. Then using the preproved theorems of the functional blocks, the HOL system could be invoked to formally verify that the design meets the specification. Once verified the scheduler might then generate the necessary hardware description and program code for the implementation. There are many avenues that could be taken, but the main intention of this work is to ensure the chosen design meets its specification.

Once this framework has been put into place further work needs to be undertaking to apply other problems of codesign, in particular the partitioning problem. As yet the methods presented have no way of representing constraints within the design, which play a major part in the determination of the final partition.

Summary

In this paper it was noted that for hardware-software codesign to be effective than the number of iterations for re-partitioning needs to be reduced. It is believed that the number of iterations required provides a measure of quality of the method used. By using formal methods to verify that a design meets a specification it is believed that a good initial partition can be chosen to reduce the number of iterations required to produce an optimum implementation. This work does not verify an implementation to be correct as this would involve using verified compilers, microprocessors, design tools, etc. It is the intention of this work to produce a framework for verifying hardware-software codesigned systems, and in the future adding constraint and communication information to consider partitioning trade-offs.

References

[1] E.W. Dijkstra (1976). A discipline of programming. Prentice-Hall.
[2] Mike Gordon (1986), "Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware." in Formal Aspects of VLSI Design, G. Milne and P.A. Subramanyam (eds.), Elsevier Science Publishers B.V., pp153-177
[3] W.A. Hunt (1987), "The Mechanical Verification of a Microprocessor Design." in HDL Descriptions to Guaranteed Correct Circuit Designs, p89-132, North-Holland.
[4] R.S. Boyer and J.S. Moore (1979), A Computational Logic, Academic Press.
[5] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," Int. Journal of Computer Simulation, vol. 4, pp. 155-182, April 1994..
[6] M.J.C. Gordon and T.F. Melham (eds) (1993), Introduction to HOL, Cambridge University Press.
[7] A. Cohn, "The Notion of Proof in Hardware Verification", Journal of Automated Reasoning, Vol. 5, May 1989, pp. 127-139
[8] P.J. Windley, "Formal Modelling and Verification of Microprocessors", IEEE Transactions on Computers, Vol. 44(1), January 1995, pp. 54-72
[9] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert and J. Van Tassel, "Experience with Embedding Hardware Description Languages in HOL." in V. Stavridou, T. F. Melham and R. T. Boute (editors), Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, Nijmegen, The Netherlands, June 1992, pages 129-156. IFIP Transactions volume A-10, North-Holland/Elsevier.
[10] K. Fisler, "A Logical Formalization of Hardware Design Diagrams." Indiana University Technical Report 416. Sept, 1994.
[11] W. Hardt and R. Camposano, "Trade-Offs in HW/SW Codesign." Presented at the International Workshop on Hardware-Software Codesign, October 7-8 1993, Cambridge, MA.