Producing design diagrams from declarative descriptions

Shaori Guo
Computing Laboratory, Oxford University, Parks Road, Oxford OX1 3QD, UK

Wayne Luk
Department of Computing, Imperial College, London SW7 2BZ, UK

ABSTRACT

The declarative language Ruby provides a coherent framework for representing and developing designs. Sketching diagrams for Ruby programs by hand is, however, time-consuming and error-prone. This paper describes a design sketcher which automates the production of a diagram from a Ruby description.

1 INTRODUCTION

Text-based languages, such as VHDL,[3] are becoming increasingly popular for developing designs. Their popularity is mainly due to their facilities for parametrising designs, and it is a great bonus if both behaviour and structure can be expressed in a single notation. Moreover, pictorial representations such as circuit schematics can be tedious to create and to modify.

Providing visual aid in hardware design is, nevertheless, important. Circuit diagrams, when appropriately drawn, make explicit the basic structure and size of components, allowing designers to obtain rapidly an overview of a design and to locate specific parts on which they can focus. There have been attempts in developing schematics with generic modules,[5] but these packages are usually confined to device-specific products.

This paper presents an approach for automatically producing design diagrams from a parametrised description in the Ruby language. Ruby is declarative: it consists of a coherent framework for representing designs as relations, and generic patterns of composing designs as functions. It has been used in developing a wide range of word-level and bit-level circuits including signal processing architectures[2] and butterfly networks,[4] and it has also been used in producing implementations partly in hardware and partly in software.[1,2]

Many users of Ruby, particularly the beginners, find diagrams an invaluable means of comprehending Ruby designs and providing intuitions for their optimisation and verification. However, sketching diagrams for Ruby programs by hand is time-consuming and error-prone. We report in this paper a method and the associated tool for sketching a Ruby design. Our sketching tool has been produced in a research project on developing a unified system for simulating, transforming, analysing and compiling declarative descriptions.[6,7,10]

There are, of course, many possible layouts for a given Ruby expression. A useful design sketcher should produce layouts that are succinct, regular, and easily understood. To meet these requirements, we have developed a sketching scheme which exploits the structure of the source program to guide its layout. The scheme allows varying component size, so that connection positions can be adapted to align the interconnecting wires either to the horizontal or to the vertical. This minimises jogs and improves comprehensibility.

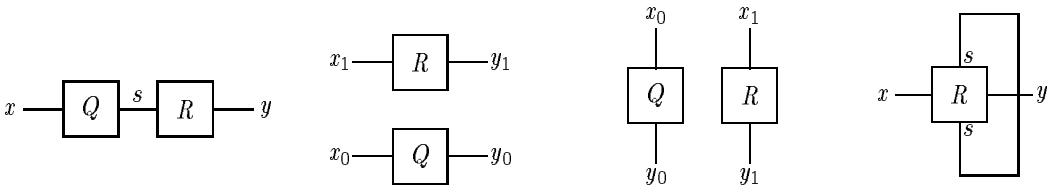
2 RUBY

Ruby is a language for describing and reasoning about designs. The background and the details of Ruby have been described elsewhere,[4,9] and only an outline will be included here.

A design is represented by a binary relation of the form $x\ R\ y$, where x, y represent the interface and belong respectively to the domain and the range of R. For instance, a squaring operation can be described by $(x\ sq\ y) \Leftrightarrow (y = x^2)$. Other examples include: $\langle x, y\rangle\ add\ z \Leftrightarrow (z = x+y)$, $\langle x, y\rangle\ max\ z \Leftrightarrow (z = maximum(x,\ y))$, $\langle x, y\rangle\ min\ z \Leftrightarrow (z = minimum(x,\ y))$. There are also wiring primitives for selecting or regrouping components of composite data, $x\ id\ y \Leftrightarrow (x = y)$, $x\ fork\ \langle y, z\rangle \Leftrightarrow (x = y \wedge x = z)$, $\langle x, y\rangle\ \pi_1\ z \Leftrightarrow (x = z)$, $\langle x, y\rangle\ \pi_2\ z \Leftrightarrow (y = z)$. Complex designs in Ruby can be formed by composing simpler designs. For instance, two components Q and R with a compatible interface can be put together by the (;) operator to form the composite design $Q\ ;\ R$ (Figure 1a): $x\ (Q\ ;\ R)\ y \Leftrightarrow (\exists s : (x\ Q\ s) \wedge (s\ R\ y))$. The $\exists$ symbol means that, unlike x and y, s is not an interface of the composite and cannot be observed.

As an example, $x\ (sq\ ;\ fork)\ y \Leftrightarrow y = \langle x^2, x^2\rangle$. We shall describe the repeated composition of n copies of Q by Q^n, so $sq^4 = sq\ ;sq\ ;sq\ ;sq$.

If there are no connections between Q and R, the composite design is represented by parallel composition $[Q, R]$, where $\langle x_0,\ x_1\rangle\ [Q, R]\ \langle y_0, y_1\rangle \Leftrightarrow (x_0\ Q\ y_0) \wedge (x_1\ R\ y_1)$. Figure 1b and Figure 1c show two ways of laying out $[Q, R]$. Note that, for example, $\langle 5, 2\rangle\ (fork\ ;\ [min, max])\ y \Leftrightarrow (y = \langle minimum(5, 2), maximum(5, 2)\rangle = \langle 2, 5\rangle)$.



a. $Q\ ;\ R$ b. $[Q, R]$ (horiz.) c. $[Q, R]$ (vert.) d. $loop\ R$

Figure 1: Some Ruby operators

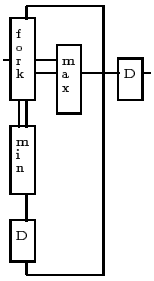
To deal with sequential designs, a relation in Ruby can be considered to relate an infinite sequence of data in its domain to another infinite sequence in its range; elements in these infinite sequences can be regarded as the values appearing at an interface at successive clock cycles. Given that $\forall t$ denotes "for all values of t", the squarer can be described by $x\ sq\ y \Leftrightarrow (\forall t : x_t^2 = y_t)$. A latch can be modelled by a delay relation D, given by $x\ D\ y \Leftrightarrow \forall t : x_{t-1} = y_t$.

Latches are also used in serial designs to prevent unbuffered loops. A design R containing an internal feedback path s can be modelled by the loop operator $loop$ (Figure 1d): $x\ (loop\ R)\ y \Leftrightarrow (\exists s : \langle x, s\rangle\ R\ \langle s, y\rangle)$.

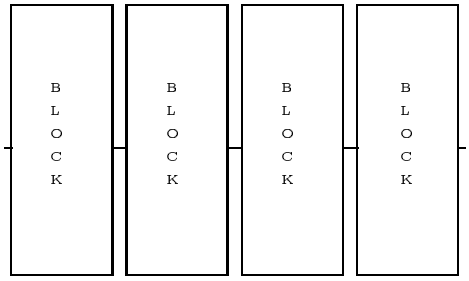
3 OVERVIEW OF THE DESIGN SKETCHER

In this section we present an overview of our design sketcher, using an insertion sorter[9] as an example. The Ruby description of our insertion sorter is as follows:

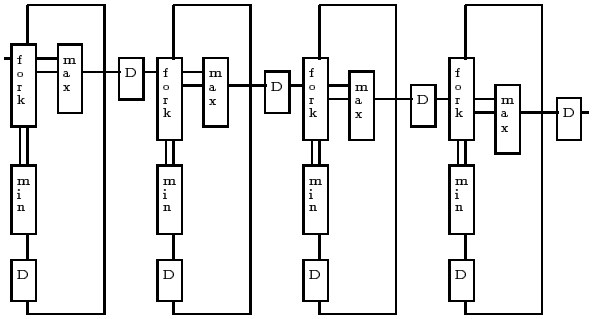
$$N \quad = \quad 4. \tag{1}$$



a. A sketch of *block*



b. A sketch of *sorter*1



c. A sketch of *sorter*2

Figure 2: Sketches of some Ruby expressions

$$cell = fork\ ;\ [min\ ; D,\ max]. \tag{2}$$

$$block = loop\ cell\ ;\ D. \tag{3}$$

$$sorter1 = (\{block\}(BLOCK))^N. \tag{4}$$

$$sorter2 = block^N. \tag{5}$$

The insertion sorter is implemented as a linear array of comparators (expression 5), and the length of the array is 4 (expression 1). Each comparator has two components: *max* for finding the maximum of two numbers and *min* for finding their minimum (expression 2). It has a feedback path where the minimum value is fed back, while the maximum value is output to the next cell (expression 3).

Currently the output of our design sketcher can be produced in LaTeX format, but it would not be difficult to provide other backends. The design sketcher includes facilities for drawing particular parts of a circuit and for producing layouts to a specified level of detail. For example, we can draw the particular part, *block*, as in Figure 2a. To avoid showing the internal structure of *block* in the diagram of the insertion sorter, we use a pair of curly braces in the Ruby expression (see expression 4) to indicate which part of the picture (in this case *block*) should be hidden. The right curly brace is followed by a pair of parentheses which enclose the name of the encapsulated part. Figure 2b is the output of our design sketcher for Ruby expression 4, which encapsulates the details of *block*.

Figure 2c is produced by expression 5, and it shows every detail of the insertion sorter; in other words, it is a flattened version of Figure 2b.

The design sketcher is written in the language SML.[11] There is a parser for converting expressions in concrete

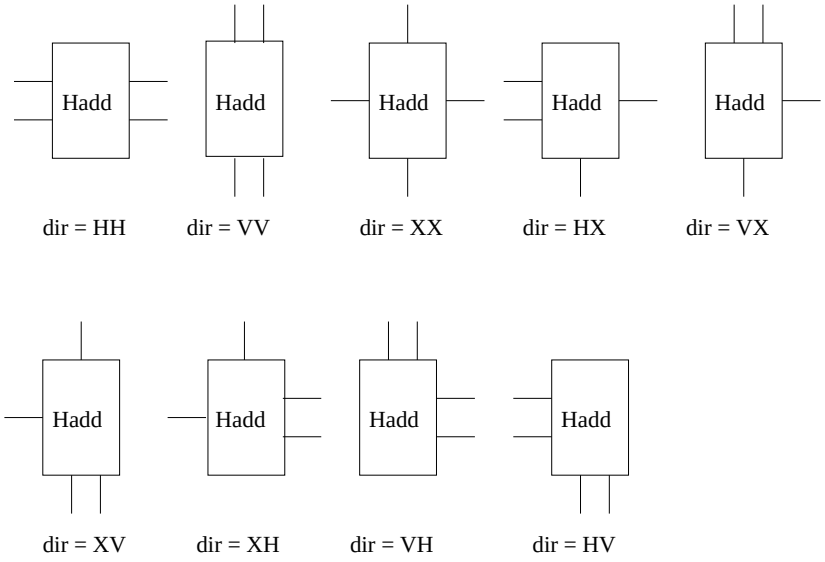


Figure 3: Nine possible layouts of a half-adder

syntax to the corresponding internal representations in abstract syntax. Other main modules of the design sketcher include: a type checker based on the unification algorithm; a mode manager which decides the level of detail the layout should be drawn, according to the source Ruby expression; a placer which produces a hierarchical placement of the layout; a sizer which adds dimensions and connection points to the description of primitive cells; a router which ensures that the connections between adjacent cells are joined together properly and, an output generator, which takes the result from the router and generates pictures in formats such as LaTeX.

The next section outlines the correspondence between a Ruby expression and its pictorial representation.

4 APPROACH

A design is represented as a rectangular block with connections on its four sides. A convention is required for assigning the domain and range data of a relation to each side of the block. The following convention is chosen: the domain data will be mapped onto the western or northern side, while the range data will be mapped onto the southern or eastern side.

Following this convention, a relation with its domain in the form of a two-tuple can be drawn as a block with some of its wires on the western side and some on the northern side, or all of them on either the western or the northern side. Similarly, a relation with its range in the form of two-tuple can be drawn as a block with some of its wires on the southern side and some on the eastern side, or all of them on either the southern or the eastern side. One can show that, for a relation with both its domain and range in the form of a two-tuple, there are nine possible layouts.

As an example, let us look at the possible layouts of a half-adder, which has two inputs and the sum and carry outputs. In Figure 3, the value of *dir* for each layout of Hadd indicates the way its domain and range are drawn, and there are nine possibilities given by HH, VV, XX, XV, XH, HX, VX, VH, HV. Each value of *dir* takes two characters: the first specifies the directional mode of the domain, while the second specifies that of the range. H means that all the connections will be drawn horizontally and V means vertical connections, and X means that both horizontal and vertical connections will be produced.

After the value of *dir* is determined, we check the compatibility of the interfaces between connected components. Since polymorphism is allowed in the domain and range of some Ruby primitives such as *fork*, a simple structure comparison is insufficient. Instead we apply a general unification algorithm to determine the most general substitution for the domain and range components, so that the interface constraints can be satisfied.

The placement and routing stages of our design sketcher are reasonably efficient because we exploit the structure of Ruby programs for placement; also the size of a block can be varied to minimise the number of jogs in the layout. This allows the rapid generation of placed and routed diagrams, although the user may not be able to control the size of components. The following describes our techniques in greater detail.

5 SPECIFICATION OF THE DESIGN SKETCHER

First, we introduce some notations which will be used in our specification. Typical parameters of a diagram consist of its dimensions, interface, and the name of the corresponding relation. To manipulate the domain and range of a Ruby expression, the functions $\mathcal{D}om$ and $\mathcal{R}ng$ are used to map a Ruby expression to its domain and range: so $\mathcal{D}om\ R$ and $\mathcal{R}ng\ R$ represent respectively the domain and range of R. The projection functions Π_1 and Π_2 extract the first and the second element of a pair, so given that the domain of R is a pair, then $\Pi_1\ (\mathcal{D}om\ R)$ will return its first element. It will also be useful to define the function *dom* and *rng* which return the domain and range component of *dir*; for instance, $dom(\ VX) = V$ and $ran(\ VX) = X$. To deal with the dimensions of a block, we define a function $\mathcal{L}$ which maps a picture P to a pair $(\mathcal{L}_w\ P, \mathcal{L}_h\ P)$ such that $\mathcal{L}_w\ P$ and $\mathcal{L}_h\ P$ are respectively the width and height of P. For instance, given a picture R with width of 3 units and height of 4 units, $\mathcal{L}_w\ R = 3$ and $\mathcal{L}_h\ R = 4$. To deal with interface, we define a function $\mathcal{I}$ which maps a picture P to a four-tuple ($\mathcal{N}\ P$, $\mathcal{E}\ P$, $\mathcal{W}\ P$, $\mathcal{S}\ P$), so that $\mathcal{N}\ P$ represents the north interface of P, $\mathcal{E}\ P$ the east interface, $\mathcal{W}\ P$ the west interface and $\mathcal{S}\ P$ the south interface. To illustrate our approach, we shall derive the layout for combinational primitives and for sequential composition.

5.1 Sketching combinational primitives

The pictorial representation of a Ruby expression is determined by its context. However, it is useful to have a default directional mode when there is no explicit context, and a library of layouts is provided in the default mode for the combinational primitives in Ruby. The context of a diagram typically contains information such as the directional mode in which it is drawn.

If we represent the diagram of a Ruby expression by enclosing it with double square brackets, the placement of a Ruby combinational primitive R can be described by

$$[\![R]\!]_{dir} \quad = \quad Atom(di,\ name)$$

where *dir* is the directional mode in which the picture would be drawn, *Atom* indicates that the picture describes a Ruby primitive, and *name* is the name for R. Information about the dimensions and the interfaces is captured by $di = (dimension,\ interface)$, where

$$\begin{array}{lcl} dimension & = & (width,\ height) \\ & = & \mathcal{L}[\![R]\!]_{dir} \\ & = & (\mathcal{L}_w\ [\![R]\!]_{dir}, \mathcal{L}_h\ [\![R]\!]_{dir}) \\ interface & = & \mathcal{I}[\![R]\!]_{dir} \\ & = & (north\_interface, east\_interface, west\_interface, south\_interface) \\ & = & (n,\ e,\ w,\ s) \end{array}$$

The values of n, e, w and s can be obtained by inspecting Figure 3. Consider the cases $dir = VV$ or VX or VH: the entire domain of R will be mapped onto the northern side of the block. If $dom(dir) = X$, then only the second component of the domain will be mapped onto the northern side. Hence n = $\mathcal{D}om\ R$ if $dom(dir) = V$, or n = $\Pi_2\ (\mathcal{D}om\ R)$ if $dom(dir) = X$, otherwise n = ϕ. Here ϕ signifies that there are no connections. The values of e, w and s can be obtained in a similar way: for example, e = $\mathcal{R}ng\ R$ if $ran(dir) = H$, or e = $\Pi_2\ (\mathcal{R}ng\ R)$ if $ran(dir) = X$, otherwise, e = ϕ.

As an example, consider laying out the component max which takes two numbers and produces the larger of the two. Given that $\mathcal{D}om\ max = \langle x, y \rangle$ and $\mathcal{R}ng\ max = z$, the diagram for max (Figure 4a) is given by

$$[\![max]\!]_{dir} \quad = \quad Atom(di,\ \text{“}max\text{”})$$

where

$$\begin{array}{lcl} di & = & (dimension,\ interface) \\ dimension & = & (width,\ height) \\ (width,\ height) & = & (\mathcal{L}_w\ [\![max]\!]_{dir},\ \mathcal{L}_h\ [\![max]\!]_{dir}) \\ interface & = & (n,\ e,\ w,\ s) \end{array}$$

Given that $dir = HH$, we have: $n = \phi$, $e = \mathcal{R}ng\ max = z$, $w = \mathcal{D}om\ max = \langle x, y \rangle$, and $s = \phi$.

5.2 Sketching sequential composition

Given that the range of A is compatible with the domain of B, the placement of a sequential composition $(A\ ;\ B)$ can be described by

$$\begin{array}{rcll} [\![A\ ;\ B]\!]_{dir} & = & Beside\ (di,\ [\![A]\!]_{dir'},\ [\![B]\!]_{dir''}) & \text{if} \quad (dir \neq VV) \\ & = & Below\ (di,\ [\![B]\!]_{VV},\ [\![A]\!]_{VV}) & \text{if} \quad (dir = VV) \end{array}$$

Here the case involving $Beside$ indicates that the composite picture is made up of two sub-pictures lying side by side. The left sub-picture is $[\![A]\!]_{dir'}$ and the right sub-picture is $[\![B]\!]_{dir''}$. The alternative placement, specified by $Below$, indicates that $[\![A]\!]_{VV}$ is placed on the top of $[\![B]\!]_{VV}$.

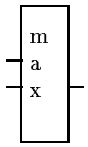
The values of dir' and dir'' are determined according to the following equations:

$$\begin{array}{rcl} dom(dir') & = & dom(dir), \\ ran(dir'') & = & ran(dir), \\ dom(dir'') & = & ran(dir'). \end{array}$$

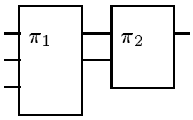
For instance if $dir = XV$, then $dir' = XH$ and $dir'' = HV$.

Let $di = (dimension,\ interface)$ and Max be a function that returns the larger of a pair of numbers. We have

$$\begin{array}{rcl} dimension & = & (width, height) \\ & = & \mathcal{L}([\![A\ ;\ B]\!]_{dir}) \\ & = & (\mathcal{L}_w\ ([\![A\ ;\ B]\!]_{dir}),\ \mathcal{L}_h\ ([\![A\ ;\ B]\!]_{dir})) \\ & = & (Max(\mathcal{L}_w\ [\![A]\!]_{dir'}, \mathcal{L}_w\ [\![B]\!]_{dir''}),\ \mathcal{L}_h\ [\![A]\!]_{dir'} + \alpha + \mathcal{L}_h\ [\![B]\!]_{dir''}),\ \text{if} \quad (dir = VV) \\ & = & (\mathcal{L}_w\ [\![A]\!]_{dir'} + \mathcal{L}_w\ [\![B]\!]_{dir''} + \beta,\ Max(\mathcal{L}_h\ [\![A]\!]_{dir'}, \mathcal{L}_h\ [\![B]\!]_{dir''})),\ \text{otherwise} \end{array}$$



a. $[\![max]\!]_{dir=HH}$



b. $[\![\pi_1 \ ; \ \pi_2]\!]_{dir=HH}$

Figure 4: Sketches of the Ruby primitive max and sequential composition of π_1 and π_2

where α and β are respectively the vertical and the horizontal extensions of the connections between A and B.

$$\begin{array}{rcl} interface & = & (\mathcal{N}\ [\![A \ ; \ B]\!]_{dir},\ \mathcal{E}\ [\![A \ ; \ B]\!]_{dir},\ \mathcal{W}\ [\![A \ ; \ B]\!]_{dir},\ \mathcal{S}\ [\![A \ ; \ B]\!]_{dir}) \\ & = & (n,\ e,\ w,\ s) \end{array}$$

where $n = \mathcal{D}om\ A$ if $dom(dir) = V$, or $n = \Pi_2\ (\mathcal{D}om\ A)$ if $dom(dir) = X$, otherwise $n = \phi$.

This follows from the rule that $dom(dir') = dom(dir)$. The values of e, w and s can be derived in a similar way:

$$\begin{array}{rcll} e & = & \mathcal{R}ng\ B & \text{if } ran(dir) = H \\ & = & \Pi_2\ (\mathcal{R}ng\ B) & \text{if } ran(dir) = X \\ & = & \phi & \text{otherwise} \\ w & = & \mathcal{D}om\ A & \text{if } dom(dir) = H \\ & = & \Pi_1\ (\mathcal{D}om\ A) & \text{if } dom(dir) = X \\ & = & \phi & \text{otherwise} \\ s & = & \mathcal{R}ng\ B & \text{if } ran(dir) = V \\ & = & \Pi_1\ (\mathcal{R}ng\ B) & \text{if } ran(dir) = X \\ & = & \phi & \text{otherwise} \end{array}$$

As an example, consider laying out $\pi_1 \ ; \ \pi_2$. Given that $dir = HH$ and $di = (dimension,\ interface)$,

$$\begin{array}{rcl} [\![\pi_1 \ ; \ \pi_2]\!]_{dir} & = & Beside(di,\ [\![\pi_1]\!]_{HH}, [\![\pi_2]\!]_{HH}) \\ dimension & = & (width,\ height) \\ & = & \mathcal{L}\ ([\![\pi_1 \ ; \ \pi_2]\!]_{HH}) \\ & = & (\mathcal{L}_w\ ([\![\pi_1 \ ; \ \pi_2]\!]_{HH}),\ \mathcal{L}_h\ ([\![\pi_1 \ ; \ \pi_2]\!]_{HH})) \\ & = & (\mathcal{L}_w\ [\![\pi_1]\!]_{HH} + \mathcal{L}_w\ [\![\pi_2]\!]_{HH} + \alpha,\ Max(\mathcal{L}_h\ [\![\pi_1]\!]_{HH}, \mathcal{L}_h\ [\![\pi_2]\!]_{HH})) \\ interface & = & (\mathcal{N}\ [\![\pi_1 \ ; \ \pi_2]\!]_{dir},\ \mathcal{E}\ [\![\pi_1 \ ; \ \pi_2]\!]_{dir},\ \mathcal{W}\ [\![\pi_1 \ ; \ \pi_2]\!]_{dir},\ \mathcal{S}\ [\![\pi_1 \ ; \ \pi_2]\!]_{dir}) \\ & = & (n,\ e,\ w,\ s) \end{array}$$

To determine n, e, w and s, let $\mathcal{D}om\ \pi_1 = \langle\langle x1, x2\rangle, y\rangle$, $\mathcal{R}ng\ \pi_2 = x2$ and $dir = HH$ (Figure 4b): we get $n = \phi$, $e = \mathcal{R}ng\ \pi_2 = x2$, $w = \mathcal{D}om\ \pi_1 = \langle\langle x1, x2\rangle, y\rangle$ and $s = \phi$.

6 FUTURE WORK

An extension of the Ruby design sketcher is to use it to guide the placement and routing of components in design synthesis; the result may then be passed to a low-level hardware compiler such as the OAL system[8] or other automatic place-and-route tools. Much of our current effort has been concentrated on the framework for mapping Ruby programs into layouts, and we have not explored optimisations of the mapping scheme. There are many possibilities of extending the design sketcher, such as including a mechanism – deterministic or otherwise – for selecting an appropriate strategy to optimally layout each expression in the Ruby program, or allowing the user to guide the layout by annotating the Ruby description.

On the theoretical side, it may be possible to recast our approach in a categorical setting.[1] Further work relating the design sketcher to source transformations[4,8] will also be useful, since Ruby offers an algebraic framework with many useful laws for optimising designs.

7 ACKNOWLEDGEMENTS

Thanks to members of the Oxford University Hardware Compilation Group for discussions and suggestions. The support of ESPRIT OMI/HORN (P7249) project, Oxford Parallel Applications Centre and Xilinx Development Corporation is gratefully acknowledged. The first author thanks the Sino-British Friendship Scholarships (SBFSS) Foundation for their support.

8 REFERENCES

[1] C. Brown and G. Hutton, "Categories, allegories and circuit design", in *IEEE Symp. on Logic in Computer Science*, July 1994.

[2] S. Guo, W. Luk and P. Probert, "Developing parallel architectures for range and image sensors", in *Proc. IEEE Int. Conf. on Robotics and Automation*, IEEE Computer Society Press, 1994, pp. 2205–2210.

[3] *IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076–1987, New York, 1988.

[4] G. Jones and M. Sheeran, "Circuit design in Ruby", in *Formal Methods for VLSI Design*, J. Staunstrup (ed.), North-Holland, 1990, pp. 13–70.

[5] S.H. Kelem and J.P. Seidel, "Shortening the design cycle for programmable logic devices", *IEEE Design and Test of Computers*, December 1992, pp. 40-50.

[6] W. Luk, G. Jones and M. Sheeran, "Computer-based tools for regular array design", in *Systolic Array Processors*, J. McCanny, J. McWhirter and E. Swartzlander (eds.), Prentice-Hall International, 1989, pp. 589–598.

[7] W. Luk, "Analysing parametrised designs by non-standard interpretation", in *Proc. Int. Conf. on Application-Specific Array Processors*, S.Y. Kung, E. Swartzlander, J.A.B. Fortes and K.W. Przytula (eds.), IEEE Computer Society Press, 1990, pp. 133–144.

[8] W. Luk and I. Page, "Parameterising designs for FPGAs", in *FPGAs*, W. Moore and W. Luk (eds.), Abingdon EE&CS Books, 1991, pp. 284–295.

[9] W. Luk, V. Lok and I. Page, "Hardware acceleration of divide-and-conquer paradigms: a case study", in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, D.A. Buell and K.L. Pocek (eds.), IEEE Computer Society Press, 1993, pp. 192–201.

[10] W. Luk, T. Cheung, Q. Miller and G. Hutton, *Simulating and Compiling Designs using Rebecca*, Technical Report, Oxford University Computing Laboratory, September 1993.

[11] L.C. Paulson, *ML for the Working Programmer*, Cambridge University Press, 1991.

[12] W. Luk and T. Wu, "Towards a declarative framework for hardware-software codesign", in *Proc. Third International Workshop on Hardware/Software Codesign*, IEEE Computer Society Press, 1994, pp. 181–188.