The T-Ruby design system

Sharp, Robin; Rasmussen, Ole

Published in:
Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95

Link to article, DOI:
10.1109/ASPDAC.1995.486374

Publication date:
1995

Document Version
Publisher's PDF, also known as Version of record

The T-Ruby Design System

Robin Sharp

Dept. of Computer Science
Technical University of Denmark
e-mail: robin@id.dtu.dk

Ole Rasmussen

Dept. of Computer Science
Technical University of Denmark
e-mail: osr@id.dtu.dk

Abstract— This paper describes the T-Ruby system for designing VLSI circuits, starting from formal specifications in which they are described in terms of relational abstractions of their behaviour. The design process involves correctness-preserving transformations based on proved equivalences between relations, together with the addition of constraints. A class of implementable relations is defined. The tool enables such relations to be simulated or translated into a circuit description in VHDL. The design process is illustrated by the derivation of a circuit for 2-dimensional convolution.

I. Introduction

This paper describes a computer-based system, known as T-Ruby [10], for designing VLSI circuits starting from a high-level, mathematical specification of their behaviour: A circuit is described by a binary relation between appropriate, possibly complex domains of values, and simple relations can be composed into more complex ones by the use of a variety of combining forms which are higher-order functions.

The basic relations and their combining forms generate an algebra, which defines equivalences (which may take the form of equalities or conditional equalities) between relational expressions. In terms of circuits, each such equivalence describes a general correctness preserving transformation for a whole family of circuits of a particular form. In the design process, these equivalences are exploited to transform a "specification" in the form of one Ruby expression to an "implementation" in the form of another Ruby expression, in a calculation-oriented style [3, 7, 11].

T-Ruby is based on a formalisation of Ruby, originally introduced by Jones and Sheeran [2], as a language of functions and relations, which we refer to as the *T-Ruby language*. The purpose of the paper is to demonstrate how such a general language can be used to bridge the gap between a purely mathematical specification and the implementable circuit. The design of a circuit for 2-dimensional convolution is used to illustrate some of the features of the method, in particular that the step from a given mathematical specification to the initial Ruby description is small and obvious, and that the method allows us to derive generic circuits where the choice of details can be postponed until the final actual synthesis.

The T-Ruby system enables the user to perform the desired transformations in the course of a design, to simulate the behaviour of the resulting relation and to translate the final Ruby description of the relation into a VHDL description of the corresponding circuit for subsequent synthesis by a high-level synthesis tool. The transformational style of design ensures the correctness of the final circuit with respect to the initial specification, assuming that the equivalences used are correct. Proofs of correctness are performed with the help of a separate theorem prover, which has a simple interface to T-Ruby, so that proof burdens can be passed to the prover and proved equivalences passed back for inclusion in T-Ruby's database.

The division of the system into the main T-Ruby-system, a theorem prover and a VHDL translator has followed a "divide and conquer" philosophy. Theorem proving can be very tedious and often needs specialists. In our system the designer can use the proved transformation rules in the computationally relatively cheap T-Ruby system, leaving proofs of specific rules and conditions to the theorem prover. When a certain level of concretisaion is reached efficient tools already exist to synthesise circuits. Therefore we have chosen to translate our relational descriptions into VHDL.

II. Ruby

The work described in this paper is based on the so-called Pure Ruby subset of Ruby, as introduced by Rossen [8]. This makes use of the observation that a very large class of the relations which are useful for describing VLSI circuits can be expressed in terms of four basic elements: two relations and two combining forms. These are usually defined in terms of synchronous streams of data as shown in Figure 1. In the figure, the type $\mathsf{sig}(\mathcal{T})$ is the type of streams of values of type $\mathcal{T}$, usually represented as a function of type $\mathbb{Z} \to \mathcal{T}$, where we identify $\mathbb{Z}$ with the *time*. The notation aRb means that a is related to b by R, and is synonymous with $(a, b) \in R$.

The relation $\mathsf{spread}\, f$ is the lifting to streams of the pointwise relation $\mathcal{R}$ of type $\alpha \sim \beta$, whose character-

$$a : \mathsf{sig}(\alpha)\ (\mathsf{spread}\, f)\ b : \mathsf{sig}(\beta) \triangleq \forall t : \mathbb{Z} \cdot (f\ a(t)\ b(t)) \quad (1)$$

$$a : \mathsf{sig}(\alpha)\ \mathcal{D}\ b : \mathsf{sig}(\alpha) \triangleq \forall t : \mathbb{Z} \cdot a(t) = b(\mathbf{Succ}(t)) \quad (2)$$

$$a : \mathsf{sig}(\alpha)\ (F\ ;\ G)\ b : \mathsf{sig}(\beta) \triangleq \exists c : \mathsf{sig}(\gamma) \cdot (a\ F\ c\ \wedge\ c\ G\ b) \quad (3)$$

$$\langle a_1 : \mathsf{sig}(\alpha_1), a_2 : \mathsf{sig}(\alpha_2) \rangle\ [F, G]\ \langle b_1 : \mathsf{sig}(\beta_1), b_2 : \mathsf{sig}(\beta_2) \rangle \triangleq a_1\ F\ b_1\ \wedge\ a_2\ G\ b_2 \quad (4)$$

Fig. 1. The basic elements of Pure Ruby

istic function is f (of type $\alpha \to \beta \to \mathbb{B}$), such that $(f\ a\ b)$ is true iff $(a, b) \in \mathcal{R}$. The type of $\mathsf{spread}\, f$ is then $\mathsf{sig}(\alpha) \sim \mathsf{sig}(\beta)$, the type of relations between streams of type α and streams of type β. For notational convenience, and to stress the idea that it describes the lifting to streams of a pointwise relation of type $\alpha \sim \beta$, this type will be denoted $\alpha \overset{sig}{\sim} \beta$.

Thus $\mathsf{spread}\, f$, for suitable f, describes (any) synchronously clocked combinational circuit, while the relation $\mathcal{D}$ – the so-called *delay* element – describes the basic sequential circuit. $F\,;\,G$ (the backward relational composition of F with G) describes the serial composition of the circuit described by F with that described by G. If F is of type $\alpha \overset{sig}{\sim} \gamma$ and G is of type $\gamma \overset{sig}{\sim} \beta$, then this is of type $\alpha \overset{sig}{\sim} \beta$. Finally, $[F, G]$ (the relational product of F and G) describes the parallel composition of F and G. For F of type $\alpha_1 \overset{sig}{\sim} \beta_1$ and G of type $\alpha_2 \overset{sig}{\sim} \beta_2$, this is of type $(\alpha_1 \times \alpha_2) \overset{sig}{\sim} (\beta_1 \times \beta_2)$. The types of the relations describe the types of the signals passing through the interface between the circuit and its environment. However, the relational description does *not* specify the direction in which data passes through the interface. "Input" and "output" can be mixed in both the domain and the range.

A feature of Ruby is that relations and combinators not only have an interpretation in terms of circuit elements, but also have a natural *graphical interpretation*, corresponding to an abstract floorplan for the circuits. The conventional graphical interpretation of spread (or of any other circuit whose internal details we do not wish to show) is as a labelled rectangular box. The components of the domain and range are drawn as wire stubs, whose number reflects the types of the relations in an obvious manner: a simple type gives a single stub, a pair type two and so on. The components of the domain are drawn up the left hand side and the range up the right. The remaining elements of Pure Ruby are drawn in an intuitively obvious way, as illustrated in Figure 2. For further details, see [2].

III. The T-Ruby Language

In T-Ruby all circuits and combinators are defined in terms of the four Pure Ruby elements using a syntax in the style of the typed lambda calculus. Definitions of some circuits and combinators with their types are given in Figure 3. α, β and so on denote type variables and can thus stand for any type. The first five definitions are of non-parameterised stream relations, which correspond to *circuits*. $+$, defined using the spread element applied to a function which evaluates to true when z equals the sum of x and y, pointwise relates two integers to their sum. ι is the (polymorphic) identity relation, dub pointwise relates a value to a pair of copies of that value and reorg pointwise relates two ways of grouping three values into pairs. These all describe *combinational circuits*; all except $+$ just describe patterns of wiring, and are known as *wiring relations*. The fifth, *SUMspec*, describes a simple *sequential circuit*: an adding machine with an accumulator register.

The remaining definitions are examples of *combinators*, which always have one or more parameters, typically describing the circuits to be combined. Applying a combinator to suitable arguments gives a circuit. Thus $(\mathsf{Fst}\, R)$ is the circuit described by $[R, \iota]$, $R \updownarrow S$ (where the combinator $\updownarrow$ is written as an infix operator) is the circuit where R is *below* S and the second component in the domain of R is connected to the first component of the range of S. In the definition of $\updownarrow$ (and elsewhere), R^{-1} denotes the inverse relation to R. The graphical interpretations of Fst and $\updownarrow$ are shown in Figure 4.

In the T-Ruby language, "repetitive" combinators and wiring relations are parameterised in the number of repetitions. This is reflected in the type system which includes *dependent product types* [4], a generalisation of normal function types, which enable us explicitly to express the size of repetive structures in the type system. For example, the combinator map (which "maps" a relation over all elements in a list of streams) has the polymorphic dependent type:

$$n : \mathsf{int} \to ((\alpha \overset{sig}{\sim} \beta) \to (\mathsf{nlist}[n]\alpha \overset{sig}{\sim} \mathsf{nlist}[n]\beta))$$

where $\mathsf{nlist}[n]\mathcal{T}$ is the type of lists of exactly n elements of type $\mathcal{T}$. Thus map is a function which takes an integer, n, and a relation of type $\alpha \overset{sig}{\sim} \beta$ as arguments and returns a relation whose type, $\mathsf{nlist}[n]\alpha \overset{sig}{\sim} \mathsf{nlist}[n]\beta$, is dependent on n, the so-called *Π-bound variable*. A full description of the T-Ruby type system can be found in [9]. The relation

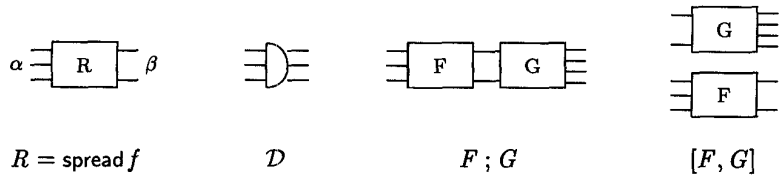

Fig. 2. Graphical interpretations of the four Pure Ruby elements

$$
\begin{array}{lcl}
+ & \triangleq & \mathsf{spread}\,(\lambda\,(x,y) : \mathsf{int} \times \mathsf{int},\ z : \mathsf{int} \cdot (z = x + y)) \\
\iota & \triangleq & \mathsf{spread}\,(\lambda\,a : \alpha,\ b : \alpha \cdot (a = b)) \\
\mathsf{dub} & \triangleq & \mathsf{spread}\,(\ \lambda\,a : \alpha,\ (b_1, b_2) : \alpha \times \alpha \cdot (a = b_1\ \wedge\ a = b_2)\ ) \\
\mathsf{reorg} & \triangleq & \mathsf{spread}\,(\ \lambda\,((a_1, a_2), a_3) : (\alpha \times \beta) \times \gamma,\ (b_1, (b_2, b_3)) : \alpha \times (\beta \times \gamma) \cdot \\
 & & \qquad (a_1 = b_1\ \wedge\ a_2 = b_2\ \wedge\ a_3 = b_3)\ ) \\
\mathit{SUMspec} & \triangleq & \mathsf{loop4}\,(\mathsf{Fst}\,(\mathsf{Snd}\,\mathcal{D})\ ;\ \mathit{ALU}_a\ ;\ (\mathsf{Snd}\,\mathsf{dub})) \\
\mathsf{Fst} & \triangleq & \lambda\,H : \alpha \overset{sig}{\sim} \beta \cdot [H, \iota] \\
\updownarrow & \triangleq & \lambda\,R : (\alpha \times \beta) \sim (\gamma \times \delta),\ S : (\epsilon \times \zeta) \sim (\beta \times \eta) \cdot \\
 & & \qquad \mathsf{reorg}\ ;\ \mathsf{Snd}\,(S)\ ;\ \mathsf{reorg}^{-1}\ ;\ \mathsf{Fst}\,(R) \\
\mathsf{map} & \triangleq & \lambda\,n : \mathsf{int},\ R : \alpha \overset{sig}{\sim} \beta \cdot \\
 & & \qquad (\mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{NNIL}\ \mathsf{else}\ ((\mathsf{apr}_{n-1})^{-1}\ ;\ [\mathsf{map}_{n-1} R, R]\ ;\ \mathsf{apr}_{n-1})\ ) \\
\mathsf{mapf} & \triangleq & \lambda\,n : \mathsf{int},\ F : (\mathsf{int} \to (\alpha \overset{sig}{\sim} \beta)) \cdot \\
 & & \qquad (\mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{NNIL}\ \mathsf{else}\ ((\mathsf{apr}_{n-1})^{-1}\ ;\ [\mathsf{mapf}_{n-1} F, (F\ n)]\ ;\ \mathsf{apr}_{n-1})\ ) \\
\mathsf{tri} & \triangleq & \lambda\,n : \mathsf{int},\ R : \alpha \overset{sig}{\sim} \alpha \cdot (\mathsf{mapf}_n (\lambda\,i : \mathsf{int} \cdot R^{i-1}) \\
\mathsf{colf} & \triangleq & \lambda\,n : \mathsf{int},\ F : (\mathsf{int} \to (\alpha \times \beta) \overset{sig}{\sim} (\beta \times \gamma)) \cdot \\
 & & \qquad (\mathsf{if}\ n = 0\ \mathsf{then}\ [\mathsf{NNIL}, \iota]\ ;\ \mathsf{cross} \\
 & & \qquad \mathsf{else}\ \mathsf{Fst}\,(\mathsf{apr}_{n-1})^{-1}\ ;\ ((\mathsf{colf}_{n-1} F) \updownarrow (F\ n))\ ;\ \mathsf{Snd}\,(\mathsf{apr}_{n-1})\ ) \\
\mathsf{rdrf} & \triangleq & \lambda\,n : \mathsf{int},\ F : (\mathsf{int} \to (\alpha \times \beta) \overset{sig}{\sim} \beta) \cdot ((\mathsf{colf}_n (\lambda\,i : \mathsf{int} \cdot (F\ i\ ;\ \pi_1^{-1})))\ ;\ \pi_1)
\end{array}
$$

Fig. 3. Examples of circuit and combinator definitions in T-Ruby

apr_n, used in the definition of map, pointwise relates an n-list of values and a single value to the $(n+1)$-list where the single value is appended "on the right" of the n-list[1].

The combinator mapf is similar to map but the second parameter is a function from integers to relations, so that the relation used can depend on its position in the structure. tri creates a triangular circuit structure, and colf a column structure where each relation is parametrised in its position in the column. Similarly, rdrf, called "reduce right" as in functional programming, if used for example with the function $(\lambda\,i : \mathsf{int} \cdot +)$ as argument, gives a relation which relates a list of integers to their sum. The graphical interpretations of some of these repetitive combinators are shown in Figure 4.

Note that the definitions are all given in a point-free notation, reflecting the fact that they are all expressed in terms of the elements of Pure Ruby. It is easy to show that they are equivalent to the expected definitions using data values; for example, that:

$$\forall\, a : \mathsf{sig}(\alpha) \cdot a\ \mathsf{dub}\ \langle a, a \rangle$$

However, defining circuits in terms of Pure Ruby elements offers several advantages: it greatly simplifies the definition and use of general rewrite rules; it simplifies reasoning about circuits in a theorem prover; and it eases the task of translating the language into a more traditional VLSI specification language such as VHDL.

IV. The Transformational Phase of T-Ruby Design

The design process in T-Ruby involves three main activities, reflecting the overall design of the system: (1) Transformation, (2) Proof and (3) Translation to

[1]Note that the size argument n here, as elsewhere, is written as a subscript to improve readibility.

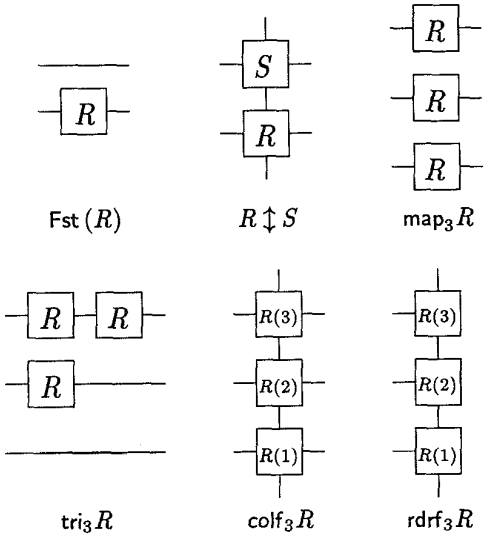

Fig. 4. Graphical interpretation of some combinators

VHDL. In this section we consider the first phase, which involves transforming an initial specification by rewriting, possibly with the addition of typing or timing constraints, so as to approach an implementable design described as a Ruby relation.

A. Rewriting with Constraints

Rewriting is an essential feature of the calculational style of design which is used in Ruby. The T-Ruby system allows the user to rewrite Ruby expressions according to pre-defined rewrite rules. Rewriting takes place in an interactive manner directed by the user, using basic rewrite functions, known as *tactics*, which can be combined by the use of higher order functions known as *tacticals*. This style of system is often called a *transformation system* to distinguish it from a conventional rewrite system. T-Ruby is implemented in the functional programming language Standard ML (SML), which offers an interactive user environment, and the tactics and tacticals are all SML functions, applied in this environment.

In the T-Ruby system, a rewrite rule is an expression with the form of an equality or an implication between two equalities, with explicit, typed universal quantification over term variables and in most cases implicit universal quantification over types via the use of type variables. Apart from this there are no restrictions on the forms of the rules which may be used. In practice, however, the commonly used rules are equalities between relational expressions, corresponding to equivalences between circuits, which can be used to manipulate a circuit description in Ruby to another, equivalent form. Rules for manipulating integer or Boolean expressions could, of course, also be introduced, but most such manipulations are performed automatically by a built-in expression simplifier based on traditional rewriting to a normal form.

Some examples of rules can be seen in Figure 5. The first rules express simple facts about the combinators, such as the commutativity of Fst and Snd (fstsndcomm), the fact that the inverse of a serial composition is the backward composition of the inverses (inversecomp), and the distributivity of Fst over serial composition (fstcompdist). The fourth rule, maptricomm, is an example of a conditional rule: the precondition that R and S commute over serial composition must be fulfilled in order for $\mathsf{tri}_n R$ and $\mathsf{map}_n S$ to commute. Similarly, forkmap states that if R is a functional relation then a single copy on the domain side of an n-way fork is equivalent to n copies on the range side of the fork. Finally, rules such as retimecol, are used in Ruby synthesis to express timing features, such as the input-output equivalence of a circuit to a systolic version of the same circuit. Note that since all these rules contain universal quantifications over relations of particular types, they essentially express general properties of whole families of circuits.

In the T-Ruby system, the directed rules used for rewriting come from three sources. They may be explicit *rewrite rule* definitions, implicit definitions derived from *circuit* or *combinator* definitions (which permit the named circuit or combinator to be replaced by its definition or vice-versa), or *lemmata* derived from previous rewrite processes which established the equality of two expressions, say t and t'.

The correctness of the explicit rules is proved by the use of a tool [6] based on the Isabelle theorem prover [5], using an axiomatisation of Ruby within ZF set theory. To make life easier for the user, conjectured rewrite rules can, however, be entered without having been proved. When rewriting is finished, all such unproved rewrite rules are printed out. Together with any instantiated conditions from the conditional rules, they form a proof obligation which the user must transfer to the theorem prover to ensure the soundness of the rewriting process.

The transformation process in the T-Ruby system primarily involves rewriting expressions as described above. However, rewriting can only produce relations which are exactly equivalent to the original, abstract specification. The T-Ruby system therefore offers the user the possibility of:

- Introducing subtyping by adding relational constraints [11].
- Modifying the timing by adding delay elements, as illustrated in the following section.
- Specialisation by instantiation of free type or term variables.

In general the transformation process starts from a relational specification, *spec*, of a circuit, at some suitably high level of abstraction. *spec* is then rewritten by a number of equality rewrites in order to reach a more imple-

$$
\begin{array}{lcl}
\mathsf{fstsndcomm} & \triangleq & \forall R : \alpha \overset{sig}{\sim} \beta,\ S : \gamma \overset{sig}{\sim} \delta \cdot (\ (\mathsf{Fst}\, R)\ ;\ (\mathsf{Snd}\, S)\ =\ (\mathsf{Snd}\, S)\ ;\ (\mathsf{Fst}\, R)\ ) \\
\mathsf{inversecomp} & \triangleq & \forall R : \alpha \overset{sig}{\sim} \beta,\ S : \beta \overset{sig}{\sim} \gamma \cdot (\ (R\ ;\ S)^{-1}\ =\ S^{-1}\ ;\ R^{-1}\ ) \\
\mathsf{fstcompdist} & \triangleq & \forall R : \alpha \overset{sig}{\sim} \beta,\ S : \beta \overset{sig}{\sim} \gamma \cdot (\ \mathsf{Fst}\,(R\ ;\ S)\ =\ (\mathsf{Fst}\, R)\ ;\ (\mathsf{Fst}\, S)\ ) \\
\mathsf{maptricomm} & \triangleq & \forall R : \alpha \overset{sig}{\sim} \alpha,\ S : \alpha \overset{sig}{\sim} \alpha,\ n : \mathsf{int} \cdot (\ (R\ ;\ S\ =\ S\ ;\ R)\ \Rightarrow \\
 & & \qquad (\mathsf{tri}_n R)\ ;\ (\mathsf{map}_n S)\ =\ (\mathsf{map}_n S)\ ;\ (\mathsf{tri}_n R)\ ) \\
\mathsf{forkmap} & \triangleq & \forall R : \alpha \overset{sig}{\sim} \alpha,\ n : \mathsf{int} \cdot (\ \mathit{is\text{-}functional}(R)\ \Rightarrow \\
 & & \qquad (R\ ;\ \mathsf{fork}_n\ =\ \mathsf{fork}_n\ ;\ (\mathsf{map}_n R))\ ) \\
\mathsf{retimecol} & \triangleq & \forall R : (\alpha \times \beta) \overset{sig}{\sim} (\beta \times \gamma),\ n : \mathit{int} \cdot (\ (\mathcal{D}\ ;\ R\ ;\ \mathcal{D}^{-1}\ =\ R)\ \Rightarrow \\
 & & \qquad (\mathsf{col}_n R\ =\ [(\mathsf{tri}_n \mathcal{D}^{-1}), (\mathcal{D}^{-1})^n]\ ;\ \mathsf{col}_n((\mathsf{Fst}\, \mathcal{D})\ ;\ R)\ ;\ \mathsf{Snd}\,(\mathsf{tri}_n \mathcal{D}))\ )
\end{array}
$$

Fig. 5. Rewrite rules in T-Ruby

mentable description. During the rewrite process the relation can be narrowed by adding relational constraints. The process can be illustrated by a series of transformations:

$$\mathit{spec} \rightarrow \mathit{step}_1 \rightarrow \mathit{step}_1' \rightarrow \mathit{step}_2 \rightarrow \mathit{step}_2'' \rightarrow \ldots \rightarrow \mathit{impl}$$

where the primes denote the added constraints. The original specification is changed accordingly from *spec* to $\mathit{spec}''^{\cdots'}$, reflecting the addition of the constraints and ensuring equality between *impl* and the final constrained specification. From a logical point of view [14], the constraints can be regarded as the assumptions under which the implementation fulfills the original specification:

$$\mathit{constraints} \vdash (\mathit{impl} \ \Leftrightarrow\ \mathit{spec})$$

B. An Example: 2-dimensional Convolution

As an example of the tranformation process, we present part of the design of a VLSI circuit for 2-dimensional discrete convolution. The mathematical definition is that from a $(2r+1)\cdot(2r+1)$ matrix $\mathcal{K}$, known as the *convolution kernel*, and a stream of values a, a new stream of values c should be evaluated, such that:

$$c(t)\ =\ \sum_{i=-r}^{+r}\sum_{j=-r}^{+r} \mathcal{K}_{i+r+1,\,j+r+1}\, a(t+wi+j)$$

The intuition behind this is that the stream a represents a sequence of rows of length w, and that each value in c is a weighted sum over the corresponding value in the a-stream and its "neighbours" out to distance $\pm r$ in two dimensions, using the weights given by the matrix $\mathcal{K}$. This is commonly used in image processing, where a is a stream of pixel values scanned row-wise from a sequence of images, and $\mathcal{K}$ describes some kind of smoothing or weighting function. Note that for each i, the summation over j is equal to the 1-dimensional convolution of a with the i'th row of $\mathcal{K}$ with a time offset of $w\cdot i$, where 1-dimensional convolution is defined by:

$$c(t) = \sum_{j=-r}^{+r} \mathcal{K}'_{j+r+1}\, a(t+j)$$

B1 Formulating the problem in Ruby

The first step in the design process is to formulate the mathematical definitions in Ruby. Following the style of design used for a correlator in [2], we now divide the relation between a and c into a combinational part, which relates c-values at a given time to a'-values at the same time (for convenience we let the summation run from 1 applying the substitution $i_{new} = i_{old} + r + 1$ and likewise for j):

$$c(t)\ =\ \sum_{i=1}^{2r+1}\sum_{j=1}^{2r+1} \mathcal{K}_{ij}\, a'_{ij}(t)$$

and a temporal part which relates the a' values at time t to the original a-values:

$$a'_{ij}(t)\ =\ a(t + w(i-r-1) + j - r - 1)$$

The temporal part, the matrix a'_{ij}, can be further split into parts which can easily be specified directly in Ruby. First we for a given i find a relation which relates b_i to a $(2r+1)$-list of a'_{ij}:

1. An offset dependent on the position j, such that $a'_{ij}(t) = a''_{ij}(t+j-1)$, which in Ruby can be specified by stating that (a'', a') are related by $(\mathsf{tri}_{2r+1}\mathcal{D}^{-1})$.
2. A $(2r+1)$-way fork, such that $a''_{ij}(t) = a'''_i(t)$, specified by $(a''', a'') \in \mathsf{fork}_{2r+1}$.
3. A fixed offset, such that $a'''_i(t) = b'_i(t-r)$, specified by $(b', a''') \in \mathcal{D}^r$.
4. Assembling 1–3 we get, for $1 \le i \le (2r+1)$

$$(b'_i, [a'_{i1}, \ldots, a'_{i,2r+1}]) \in (\mathcal{D}^r\ ;\ \mathsf{fork}_{2r+1}\ ;\ \mathsf{tri}_{2r+1}\mathcal{D}^{-1})$$

Next we find a relation relating a to a $(2r+1)$-list of b_i's:

5. An offset dependent on position i, such that $b'_i(t) = b''_i(t + w(i-1))$, specified by $(b'', b') \in \mathsf{tri}_{2r+1}(\mathcal{D}^{-1})^w$.
6. A $(2r+1)$-way fork, such that $b''_i(t) = b'''(t)$, specified by $(b''', b'') \in \mathsf{fork}_{2r+1}$.
7. Another fixed offset, such that $b'''(t) = a(t - wr)$, specified by $(a, b''') \in (\mathcal{D}^w)^r$.
8. Assembling 5–7 we get:

$$(a, [b'_1, \ldots, b'_{2r+1}]) \in ((\mathcal{D}^w)^r \,;\, \mathsf{fork}_{2r+1} \,;\, \mathsf{tri}_{2r+1}(\mathcal{D}^w)^{-1})$$

It is convenient to rewrite relations (4) and (8) above as follows:

$$(b'_i, [a'_{i1}, \ldots, a'_{i,2r+1}]) \in (\mathsf{fork}_{2r+1} \,;\, \mathsf{butterfly}_r \mathcal{D}) \quad (5)$$
$$(a, [b'_1, \ldots, b'_{2r+1}]) \in (\mathsf{fork}_{2r+1} \,;\, \mathsf{butterfly}_r \mathcal{D}^w) \quad (6)$$

where the combinator butterfly is defined by:

$$\mathsf{butterfly}\colon n : \mathsf{int} \to (\alpha \overset{sig}{\sim} \alpha) \to (\mathsf{nlist}[2n+1]\alpha \overset{sig}{\sim} \mathsf{nlist}[2n+1]\alpha)$$
$$\mathsf{butterfly} \triangleq \lambda\, n : \mathsf{int},\ R : \alpha \overset{sig}{\sim} \alpha \cdot (\mathsf{app}_{n+1,n}{}^{-1} \,;\, \mathsf{Fst}\,(\mathsf{irt}_{n+1} R^{-1} \,;\, \mathsf{apr}_n{}^{-1}) \,;\, \mathsf{reorg} \,;\, \mathsf{Snd}\,(\mathsf{apl}_n \,;\, \mathsf{tri}_{n+1} R) \,;\, \mathsf{app}_{n,n+1})$$

The combinational part of the convolution relation is easily expressed in Ruby in terms of a combinator Q, of type $\mathsf{int} \to \mathsf{int} \to ((\mathsf{int} \times \mathsf{int}) \overset{sig}{\sim} \mathsf{int})$, such that $(Q\ i\ j)$ relates (a, x) to $(\mathcal{K}_{ij}\, a + x)$, which expresses the convolution kernel as a function of position within the matrix $\mathcal{K}$. If we then define $c_i(t) = \sum_{j=1}^{2r+1} \mathcal{K}_{ij}\, a'_{ij}(t)$, it is easy to demonstrate that, for all t and arbitrary x, $([a'_{i1}(t), \ldots, a'_{i,2r+1}(t)], x(t))$ is related to $c_i(t) + x(t)$ by the Ruby relation $(\mathsf{rdrf}_{2r+1}(Q\ i))$, where rdrf is defined in Figure 3. Combining this with the temporal relations given in definitions 5 and 6 we find that the entire 2-dimensional convolution relation CR_2, which relates (a, x) and $(c + x)$ for a given w, r, x and Q can be expressed in terms of the one-dimensional convolution relation $(CR_1\ i)$, which relates (b'_i, x) and $(c_i + x)$ for given i, as follows:

$$CR_1 \triangleq \lambda\, i : \mathsf{int} \cdot (\mathsf{Fst}\,(\mathsf{fork}_{2r+1} \,;\, \mathsf{butterfly}_r \mathcal{D}) \,;\, \mathsf{rdrf}_{2r+1}\ (Q\ i))$$
$$CR_2 \triangleq \mathsf{Fst}\,(\mathsf{fork}_{2r+1} \,;\, \mathsf{butterfly}_r \mathcal{D}^w) \,;\, \mathsf{rdrf}_{2r+1}\ CR_1$$

CR_1 corresponds to the inner summation over j in the specification. The graphical interpretation of CR_2 for $r = 2$ is shown on the left in Figure 6, and the interpretation of $(CR_1\ i)$ for $i = 1$ on the right. The butterflies contain increasing numbers of delay elements, $\mathcal{D}$, above the mid-line and increasing numbers of "anti-delay" elements, $\mathcal{D}^{-1}$ below the mid-line. As follows from the definitions, the small butterflies use single delay elements, corresponding to the time difference between consecutive elements in the data stream, while the large butterflies use groups of w delay elements, corresponding to the time difference between consecutive lines in the data stream.

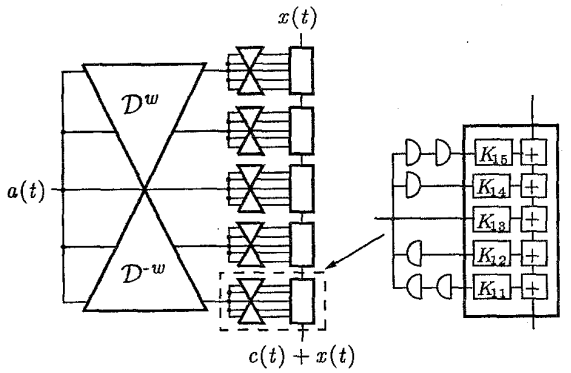

Fig. 6. Two dimensional convolution for $r = 2$

To define these relations in T-Ruby, it is convenient to parameterise them, so that they become combinators dependent on r, w and Q. The final definitions are:

$$conv1 \triangleq \lambda\, r : \mathsf{int},\ C : \mathsf{int} \to (\mathsf{int} \times \mathsf{int}) \overset{sig}{\sim} \mathsf{int} \cdot (\mathsf{Fst}\,(\mathsf{fork}_{2r+1} \,;\, \mathsf{butterfly}_r \mathcal{D}) \,;\, \mathsf{rdrf}_{2r+1}\, C)$$
$$conv2 \triangleq \lambda\, r : \mathsf{int},\ w : \mathsf{int},\ C : \mathsf{int} \to \mathsf{int} \to (\mathsf{int} \times \mathsf{int}) \overset{sig}{\sim} \mathsf{int} \cdot (\mathsf{Fst}\,(\mathsf{fork}_{2r+1} \,;\, \mathsf{butterfly}_r(\mathcal{D}^w)) \,;\, \mathsf{rdrf}_{2r+1}(\lambda\, i : \mathsf{int} \cdot (conv1\ r\ (C\ i))))$$

With these definitions, the actual circuit for 2-dimensional convolution is described by the relation $(conv2\ r\ w\ Q)$ for suitable values of r, w and Q.

B2 Transformation to an implementable relation

Unfortunately, the relation given above does not describe a physically implementable circuit, if we assume (as we implicitly have done until now) that the inputs appear in the domain of the relation (as x and a) and the outputs in the range (as c). This is because of the "anti-delays", $\mathcal{D}^{-1}$, in the butterflies. So instead of trying to implement the relation $(conv2\ r\ w\ Q)$ as it stands, we implement a retimed version of it, formed by adding a constraint on the domain side which delays all the input signals:

$$\mathsf{Fst}\,(\mathcal{D}^r \,;\, (\mathcal{D}^w)^r) \,;\, (conv2\ r\ w\ Q)$$

This will result in the anti-delays being cancelled out, as the delay elements in the constraint are moved "inwards" into the original relation. The resulting circuit will produce its outputs $r \cdot (w+1)$ time units later than the original circuit, but this is the best we can achieve in the physical world we live in!

From here on we use a series of rewrite rules to manipulate the relation into a more obviously implementable form. The derivation, shown in full in [12], finishes with the relational expression:

$$\mathsf{Fst}\,(\mathsf{fork}_{2r+1}); \mathsf{rdrf}_{2r+1}(\lambda\, k : \mathsf{int} \cdot (\mathsf{Fst}\,(\mathsf{fork}_{2r+1}) \,;\, \mathsf{Snd}\,\mathcal{D}^{w-(2r+1)}; \mathsf{rdrf}_{2r+1}(\lambda\, i : \mathsf{int} \cdot (\mathsf{Snd}\,\mathcal{D} \,;\, (Q\ k\ i)))))$$

This is a generic description of a convolution circuit, expressed in terms of three free variables: r, w and Q, corresponding respectively to the kernel size for the convolution, the line size for the 2-dimensional array of points to be convoluted, and the kernel function.

To obtain a description of a particular concrete circuit, we can then use the T-Ruby system's facilities for *instantiating* such free variables to particular values. For example, we might instantiate r to 2, w to 64 and Q to $\lambda\, i, j : \mathsf{int} \cdot (acc(i+j))$, where:

$$acc \triangleq \lambda\, W : \mathsf{int} \cdot (\ \mathsf{spread}\,(\lambda\,(m,s) : (\mathsf{int} \times \mathsf{int}),\ o : \mathsf{int} \cdot (o = s + (W * m)))\ )$$

Thus $acc(i+j)$ describes a multiply-and-add circuit with multiplication factor $(i+j)$, and the kernel element described by $(Q\ i\ j)$ will use this factor as the weight in accumulating the weighted sum.

After suitable reduction of the integer expressions, this would give us the relational description:

$$\begin{array}{l}\mathsf{Fst}\,(\mathsf{fork}_5);\\ \mathsf{rdrf}_5(\lambda\, k : \mathsf{int} \cdot (\mathsf{Fst}\,(\mathsf{fork}_5)\ ;\ \mathsf{Snd}\,\mathcal{D}^{59};\\ \qquad\qquad \mathsf{rdrf}_5(\lambda\, i : \mathsf{int} \cdot (\mathsf{Snd}\,\mathcal{D};\\ \qquad\qquad\qquad ((\lambda\, i,j : \mathsf{int} \cdot (acc(i+j)))\\ \qquad\qquad\qquad\quad k\ i)))))\end{array}$$

with no free variables. The graphical interpretation of this final version of the circuit is shown in Figure 7. As can be seen in the figure, the circuit is semi-systolic, with a latch (described by a delay element, $\mathcal{D}$) associated with each combinational element, but with a global distribution of the input stream a to all of the combinational elements.

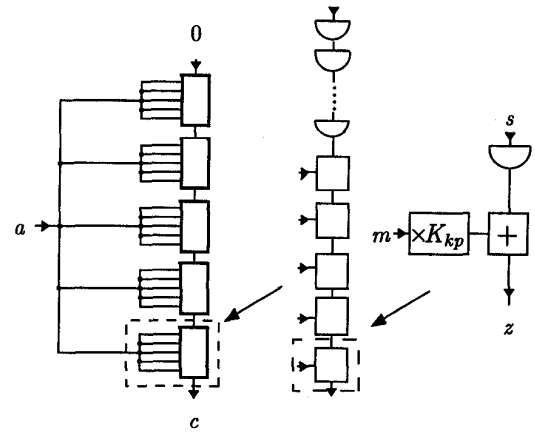

Fig. 7. Semi-systolic version of two dimensional convolution for $r = 2$

The left-hand structure depicts the entire circuit.

The basic building element shown on the right corresponds to the relation $\mathsf{Snd}\,\mathcal{D}\ ;\ (Q\ k\ p)$ with Q instantiated as described in the text.

The middle structure depicts $\mathsf{Snd}\,\mathcal{D}^{59}\ ;\ \mathsf{rdrf}_5(\lambda\, p \cdot (\mathsf{Snd}\,\mathcal{D}\ ;\ (Q\ k\ p)))$. Only 3 of the 59 delay elements in $\mathcal{D}^{59}$ are shown.

Arrows in the figure indicate the input/output partitioning determined by the causality analysis.

C. Selection and Extraction

The rewriting system of T-Ruby includes facilities for *selection* of subterms from the target expression by matching against a pattern with free variables. This can be used to restrict rewriting temporarily to a particular subterm, or, more importantly, for *extraction* of part of the target expression for implementation. In the latter case, the remainder of the target expression gives a *context* describing a set of implementation conditions that must be fulfilled for the extracted part to work

Extraction is in many respects the converse of adding relational constraints to the specification, and the context specifies the same sorts of requirement. Firstly, it may give representation rules which must be obeyed at the interface to the extracted subterm, and secondly (if the context contains delay elements, $\mathcal{D}$), it may give timing requirements for the implementation of the subterm.

V. VLSI Implementation

The relational approach to describing VLSI circuits offers a greater degree of abstraction than descriptions using functions alone, since the direction of data flow is not specified. However, real circuits offer particular patterns of data flow, and this means that the interpretation of a relation may in general be 0, 1 or many different circuits. In the case of zero circuits, we say the relation is unimplementable. The widest class of relations which are generally implementable is believed to be the *causal relations*, as defined by Hutton [1]. These generalise functional relations in the sense that inputs are not restricted to the domain nor outputs to the range.

In T-Ruby, causality analysis is performed at the end of the rewriting process, when the user has extracted the part of the relation which is to be implemented. In most cases, in fact, the context from which the relation is extracted is non-implementable: for example, it may specify timing requirements which (if they could be implemented) would correspond to foreseeing the future.

A. Causality analysis

More exactly, a relation is causal if the elements in each tuple of values in the relation can be partitioned into two classes, such that the first class (the *outputs*) are functionally determined by the second class (the *inputs*), and such that the same partitioning and functional dependency are

used for all tuples in the relation. For example, the previously defined relation $+$ is causal, in the sense that the three elements $((x, y), z)$ of each tuple of values in the relation can be partitioned as described, in fact in three different ways:

1. With x and y as inputs and z as output, so that the relation describes an *adder*.
2. With x and z as inputs and y as output, so that the relation describes a *subtractor*.
3. With y and z as inputs and x as output, so that the relation describes another *subtractor*.

Note that the relation $+^{-1}$ is also causal, although it is not functional. Essentially, causality means that the relation can be viewed under the partitioning as a deterministic function of its inputs.

In T-Ruby, the relation to be analysed is first expanded, using the definitions of its component relations, to a form where it is expressed entirely in terms of the four elements of Pure Ruby and relational inverse. The expanded relation is then analysed with a simple bottom-up analysis heuristic. For combinational elements described by spread relations, causality is determined by analysing the body of the spread, which must have the form of a *body part* which is:

- an equality with a single variable on the left-hand side,
- a conjunction of body parts, or
- a conditional choice between two body parts.

In each equality, the result of the analysis depends on the form of the right-hand side. If this is a single variable, no conclusions are drawn, as the equality then just implies a wire in the abstract floorplan. If the right-hand side is an expression, all values in it are taken to be inputs, and the left-hand side is taken to be an output. In choices, all values in the condition are taken to be inputs. If these rules result in conflicts, no causal partitioning can be found. When there are several possible causal partitionings, as in the case of $+$, on the other hand, the rules enable us to choose a unique one.

For delay elements, $\mathcal{D}$, values in the domain are inputs and those in the range are outputs. Parallel composition preserves causality, and so in fact does inversion, but serial compositions in general require further analysis, to determine whether the input/output partitionings for the component relations are compatible with an implementable (unidirectional) data flow between the components. Essentially, checks are made as to whether two or more outputs are used to assign a new signal value to the same wire, whether some wires are not assigned signal values at all or whether there are loops containing purely combinational components. This additional analysis is exploited in order to determine the network of the circuit in the form of a netlist with named wires between active components. At present there is no backtracking, so if the arbitrary choice of partitioning when there are several possibilities is the "wrong" one, then it will not be possible to find a complete causal partitioning for the entire circuit.

As an example, let us consider the analysis of parts of the relation for 2-dimensional convolution. The central element in this is the relation given by $acc(p + k)$, which describes the combinational multiply-and-add circuit for kernel element (p, k). Using the definition of acc, and substituting $(p + k)$ for W, this reduces to:

$$\mathsf{spread}\,(\lambda\,(m, s) : (\mathsf{int} \times \mathsf{int}),\ z : \mathsf{int} \cdot (z = s + (p + k) * m))$$

The body of the spread has the form of an equality with a single left-hand side, and thus the causal partitioning will make z an output and m and s both inputs. In this case, the relation is functional from domain to range, but in general this need not be so.

Since delay elements, $\mathcal{D}$, can only have inputs on the domain side and outputs on the range side, the serial composition $(\mathsf{Snd}\,\mathcal{D}\,;\,acc(p + k))$ is compatible with this analysis of $acc(p + k)$, as the range of the delay element corresponds to component s in the domain of $acc(p + k)$. Further analysis proceeds in a similar manner, leading to the final data flow pattern shown by arrows in Figure 7.

B. Translation to VHDL

Since causality analysis gives both the network of the circuit and the direction of data flow along the individual wires between components, the actual translation to VHDL is comparatively simple. Each translated "top level" Ruby relation is declared as a single design unit, incorporating a single entity with a name specified by the user. In rough terms, each combinational relation $\mathcal{C}$ which is not a wiring relation within the expanded Ruby relation is translated into one or more possibly conditional signal assignments, where the outputs of $\mathcal{C}$ are assigned new values based on the inputs. For example, the relation $acc(p + k)$ considered above gives rise to a single concurrent signal assignment of the form:

```
sig_z  <= (sig_s  + (W  * sig_m));
```

where `sig_z`, `sig_s` and `sig_m` are the names of the VHDL signals corresponding to z, s and m respectively, and `W` is a constant equal to the value of $(p + k)$ for the circuit element in question. Since the operators available for use with operands of integer, Boolean, bit and character types in Ruby are (with one simple exception: logical implication) a subset of those available in VHDL, this direct style of translation is problem-free. In a similar manner, any conditional (if-then-else) expressions in the body of a spread are directly translated into conditional assignment statements, possibly with extra signal assignments to evaluate a single signal giving the condition.

The VHDL types for the signals involved are derived from the Ruby types used in the domain and range of $\mathcal{C}$ in an obvious way. Thus for the elementary types, the Ruby type bit is translated to the VHDL type `rubybit`, bool to

rubybool, int to rubyint, and char to rubychar, where the VHDL definitions of rubybit, rubybool, rubyint and rubychar are predefined in a package RUBYDEF, which is referred to by all generated VHDL units. Composed types give rise to groups of signals, generated by (possibly recursive) flattening of the Ruby type, such that a pair is flattened into its two components, a list into its n components and so on until elementary types are reached.

If the Ruby relation refers to elementary types other than these pre-defined ones, a package declaration containing suitable type definitions is generated by the translator. For example, if an enumerated type etyp is used, a definition of a VHDL enumerated type rubyetyp with the same named elements is generated.

Free variables of relational type and all non-combinational relations in the Ruby relation are translated into instantiations of one or more VHDL *components*. For example, a Delay relation, $\mathcal{D}$. of Ruby type $t \overset{sig}{\sim} t$, where t is a simple type, will be translated into an instantiation of the component dff_rubyt, where rubyt is the VHDL type corresponding to t, as above. For composed types, such as pairs and lists, two or more components, each of the appropriate simple type, are used. Standard definitions of these components for all standard simple Ruby types are available in a library. Other components (in particular those generated from free relational variables) are assumed to be defined by the user.

The final result of translating the fully instantiated 2-dimensional convolution relation into VHDL is shown in Figure 8. The figure does not show the entire VHDL code (which of course is very repetitive owing to the regular nature of the circuit), but illustrates the style. Signal identifiers starting with input and output correspond to the external inputs and outputs mentioned in the formal port clause of the entity, while names starting with wire identify internal signals. A clock input is generated if any of the underlying entities are sequential. The assignments marked *Calculations* describe the combinational components, and those marked *Registers* describe the component instantiations corresponding to the Delay elements. (Instantiations of any other user-defined components follow in a separate section if required.)

The correctness of the translation relies heavily on two facts:

1. There is a simple mapping between Ruby types and operators and types and operators which are available in VHDL.
2. Relations are only considered translatable if an (internally consistent) causal partitioning can be found.

These facts also imply that the VHDL code which is generated can be synthesised into VLSI. At present, we use the Synopsys VHDL Compiler [13] for performing this synthesis automatically.

```
-- conv264.vhd ---- Machine generated code. Do not edit.
-- Compiled 950201, 11:58:28 from Ruby relation:
--%% ((Fst (fork 5));
--%%  (rdrf 5 \k:int.
--%%           ((((Fst (fork 5));(Snd D^59));
--%%             (rdrf 5 \p:int.(((Snd D);(acc(p+k))))

USE WORK.rubydef.ALL;

ENTITY conv264 IS
  PORT
    ( input1,input2: IN rubyint;  output1: OUT rubyint;
      clk: IN Bit );
  END conv264;

ARCHITECTURE ruby OF conv264 IS
  COMPONENT dff_rubyint
    PORT
      ( d: IN rubyint;  q: OUT rubyint;  clk: IN Bit );
    END COMPONENT;
  SIGNAL
    wire8,wire1897,wire1983,wire2105,wire2290,wire2435,
      wire2540,wire4546,wire4548,wire4550,wire4552,
        ...
      wire19943,wire20128,wire20273,wire20378,wire20478,
      wire20596,wire20746,wire20928,wire21142: rubyint;
  BEGIN
    -- Input assignments: --
    wire8 <= input1;
    wire1897 <= input2;
    -- Output assignments: --
    output1 <= wire1983;
    -- Calculations: --
    wire1983  <= (wire5406  + (2  * wire8));
    wire5306  <= (wire5524  + (3  * wire8));
    wire5201  <= (wire5674  + (4  * wire8));
         ...
    wire19943 <= (wire21142 + (10 * wire8));
    -- Registers: --
    D1: dff_rubyint PORT MAP (wire19734,wire21142,clk);
    D2: dff_rubyint PORT MAP (wire19943,wire20928,clk);
    D3: dff_rubyint PORT MAP (wire20128,wire20746,clk);
    D4: dff_rubyint PORT MAP (wire20273,wire20596,clk);
    D5: dff_rubyint PORT MAP (wire20378,wire20478,clk);
          ...
    D319: dff_rubyint PORT MAP (wire4546,wire4548,clk);
    D320: dff_rubyint PORT MAP (wire2540,wire4546,clk);
  END ruby;
```

Fig. 8. VHDL translation of the instantiated 2-dimensional convolution circuit

C. Other Components of the System

The complete system is illustrated in Figure 9. A similar style of analysis to that used for generating VHDL code is used for controlling simulation of the behaviour of the extracted relation. The user must supply a stream of values for the inputs of the circuit and, if required, initial values for the latches, and the simulation then uses exactly the same assignments of new values to signals as appear in the VHDL description. Obviously, only fully instantiated causal relations can be simulated.

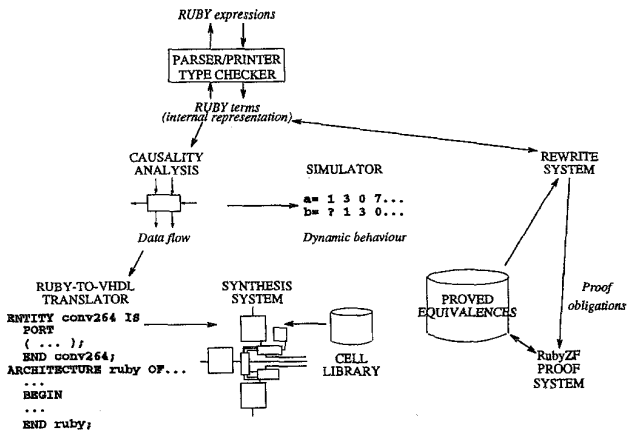

Fig. 9. The complete Ruby Design System

VI. Conclusion

In this paper, we have presented the T-Ruby Design System and outlined a general design method for VLSI circuits based on transformation of formal specifications using equality rewriting, constraints and extraction. The simple mathematical basis of the specification language in terms of functions and relations enables us to prove general transformation rules, and minimises the step from the mathematical description of the problem to the initial specification in our system.

The use of the system has been illustrated by the non-trivial example of a circuit for 2-dimensional convolution. This example shows how T-Ruby can be used to describe complex repetitive structures which are useful in VLSI design, and demonstrates how the system can be used to derive descriptions of highly generic circuits, from which concrete circuit descriptions can be obtained by instantiation of free parameters. Circuits described by so-called causal relations can be implemented and their behaviour simulated. In the T-Ruby system, a simple mapping from T-Ruby to VHDL for such relations is used to produce a VHDL description for final synthesis.

The design system basically relies on the existence of a large database of pre-proved transformation rules. However, during the design process, conjectured rules can be introduced at any time, and rewrite rules with pre-conditions may be used. The system keeps track of the relevant proof burdens and these can be transferred later to a separate theorem prover. Our belief is that this "divide and conquer" philosophy helps to make the use of formal methods more feasible for practical designs.

VII. Acknowledgements

The work described in this paper has been partially supported by the Danish Technical Research Council.

The authors would like to thank Ole Sandum for his work on the design of the Ruby to VHDL translator.

References

[1] Graham Hutton. *Between Functions and Relations in Calculating Programs.* PhD thesis, Department of Computer Science, University of Glasgow, 1993.

[2] G. Jones and M. Sheeran. Circuit design in Ruby. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design*, pages 13–70. Elsevier, 1990.

[3] G. Jones and M. Sheeran. Relations and refinement in circuit design. In C. C. Morgan and J. C. P. Woodcock, editors, *Proceedings of the 3rd. BCS FACS Workshop on Refinement*, Workshops in Computing, pages 133–152, 1991. BCS, Springer-Verlag.

[4] P. Martin-Löf. Constructive mathematics and computer programming. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 167–184. Prentice-Hall, London, 1985.

[5] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, LNCS volume 828. Springer-Verlag, 1994.

[6] O. Rasmussen. A Ruby proof system. Technical Report ID–TR: 1995-xx, Dept. of Computer Science, Technical University of Denmark, 1995. To appear.

[7] L. Rossen. Proving (facts about) Ruby. In G. Birtwhistle, editor, *IV Higher Order Workshop, Banff*, Workshops in Computing, pages 265–283. Springer-Verlag, 1990.

[8] L. Rossen. Ruby algebra. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits, Oxford 1990*, Workshops in Computing, pages 297–312. Springer-Verlag, 1990.

[9] R. Sharp. The Ruby framework. Technical Report ID–TR: 1993-119, Dept. of Computer Science, Technical University of Denmark, June 1993.

[10] R. Sharp. T-Ruby: A tool for handling Ruby expressions. Technical Report ID–TR: 1994-154, Dept. of Computer Science, Technical University of Denmark, December 1994.

[11] R. Sharp and O. Rasmussen. Transformational rewriting with Ruby. In L. Claesen, editor, *CHDL'93*, pages 243–260. IFIP WG10.2, Elsevier, 1993.

[12] R. Sharp and O. Rasmussen. Using a language of functions and relations for VLSI specification. In *Functional Programming and Computer Architecture, FPCA'95*, 1995. To appear.

[13] Synopsys, Inc. *VHDL Compiler Reference Manual*, 3.1 edition, 1994.

[14] D. Weise. Constraints, abstraction and verification. In M. Leeser and G. Brown, editors, *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, LNCS volume 408, pages 25–39. Springer-Verlag, 1989.