Gropius - Advanced Reuse Concepts in a New Hardware Description Language [*]

Dirk Eisenbiegler, Christian Blumenröhr
Institute for Circuit Design and Fault Tolerance (Prof. Dr.-Ing. D. Schmid),
University of Karlsruhe, Germany e–mail: {eisen,blumen}@ira.uka.de

Abstract

In this paper we present a new hardware description language named Gropius. Gropius ranges from the gate level to the system level and supports abstraction and design reuse in a systematic manner. Gropius was designed for a formal synthesis scenario, where synthesis is performed by applying mathematical derivation steps, thus guaranteeing correctness of the synthesis process. Since Gropius was defined in a mathematical manner, its semantics is precise and unambiguous.

1 Introduction

In circuit design more and more abstract design levels are used in order to manage the complexity of nowadays systems. Synthesis means mapping an abstract circuit description given by the designer to a concrete implementation in the real world. In synthesis, hardware description languages play an important role. They determine the set of circuit descriptions to be considered as input and output of some synthesis step. A synthesis concept is always closely related to specific hardware description languages. This is why designing a good hardware description language is that important.

So what are the main objectives when designing a real good hardware description language? We believe, that there are three major points that have to be considered:

expressiveness To get along with the complexity of large circuits, one has to go beyond pure gate level net lists. More abstract levels of circuit descriptions are required: rt-level, algorithmic level, system level. Among other features,

[*]This work has been partly financed by the Deutsche Forschungsgemeinschaft, Project SCHM 623/6-1.

hardware description languages at these levels have to support abstract data types as well as abstract timing descriptions. The features provided by the hardware description language must correspond to a design concept that allows the circuit designer to represent circuits in a compact manner. Providing a good concept for design reuse is a major matter as to the quality of a hardware description language.

unambiguous semantics It seems to be clear, that it is necessary to explain what the syntactical elements of a hardware description language stand for. Good hardware description languages are always related to a simple semantics that can easily be described in a mathematical notation. A hardware description language with a complicated semantics based on an awkward timing model leads to design errors due to misunderstandings.

minimum size A hardware description language should be as small as possible. This means strictly eliminating redundant constructs. Both the size of a hardware description language and the complexity of the semantics determine the costs for teaching it to new circuit designers.

We believe that existing hardware description languages such as VHDL or Verilog are far worse than they could be. Most problems of nowadays hardware description languages can be illustrated with VHDL. VHDL is one of the most widespread hardware description languages. In some sense VHDL is very expressive: the set of systems one can describe with VHDL goes beyond what may be built in reality. Therefore, all synthesis tools have to reduce themselves to subsets of VHDL. There is an informal semantics for VHDL [VHDL96], however it is not unambiguous. Several attempts have been made to give VHDL a formal, precise semantics, however, there is no common standard [KlBr95]. As a result, different tools interprete VHDL sources in different ways. In general, tools with VHDL interfaces are not compatible, designs cannot be reused and it is not possible to process VHDL code from one tool to the next. Besides the semantics problem of VHDL, there is also a lack as to the expressiveness of the language. The design concepts of VHDL are pretty poor compared to modern programming languages in software design. Several attempts have been made to extend VHDL by modern design/reuse concepts [RaPN97, SwMC95].

2 Gropius — a survey

This article presents a new hardware description language named Gropius [*]. Gropius has an exact, unambiguous mathematical semantics. It was designed for a formal synthesis tool, where synthesis is performed within a theorem prover system by applying mathematical rules (see [KBES96] for a survey on formal

[*] WALTER GROPIUS (1883-1969), founder of the BAUHAUS (*form follows function*).

synthesis). All constructs of Gropius have been defined within logic — in other terms: Gropius is nothing but a subset of higher order logic. Gropius is pretty expressive and supports different techniques related to design reuse. We will first briefly introduce Gropius and then explain the techniques for design reuse.

Gropius is a functional hardware description language. It provides means for the user to define arbitrary functions. User defined functions are the common basis for hardware descriptions at different levels of abstraction (see section 3). Gropius is strongly-typed and, unlike Ella [MoCl93], includes an automatic type inference mechanism. Gropius supports polymorphic and generic structures and allows parameterizing circuits with subcomponents. These are very powerful means for a systematic reuse of designs. Unlike HML [LLLA93], Ruby [ShRa95], DDD [JoBo91] or Lustre [HCRP91], Gropius is not restricted to lower levels of abstractions, but also supports circuit descriptions at the algorithmic and system level.

Gropius is a very small language with a minimum number of constructs. There are only 11 syntax rules (see figure 1). Also the number of basic key words is pretty small: there are 5 basic boolean operators, 25 abstract operators for non-boolean expressions (polymorphism, enumeration types, arrays etc.), 9 interface patterns and 7 different k-processes (see figure 2). The small size makes it easy for the user to learn the language. In VHDL for example, there are more than 150 syntax rules (see [VHDL96]).

Gropius designed to support a specific design methodology. Design reuse is one of its strengths. In this paper, we will give a brief introduction to Gropius and describe some major concepts and their significance as to design reuse.

3 Design reuse across abstraction levels

In Gropius, all functions are based on a fixed set of elementary boolean and abstract operators (see figure 3). User defined functions are subdivided in three groups: boolean dfg-terms, dfg-terms and p-terms. Boolean dfg-terms are a subset of dfg-terms and dfg-terms are a subset of p-terms. The set of p-terms covers the entire set of computable functions. Dfg-Terms are non-recursive compositions of elementary operators. Boolean dfg-terms are exclusively build on basic boolean operators.

Unlike most standard hardware description languages such as VHDL and Verilog, Gropius is strictly divided into sublanguages each corresponding to a specific abstraction level: Gropius-0 — gate level, Gropius-1 — rt-level, Gropius-2 — algorithmic level, Gropius-3 — system level. Due to the common core of the sublanguages, Gropius is not just a set of hardware description languages. The elementary constructs and the user defined functions are a common starting point for all circuit descriptions.

Boolean dfg-terms and dfg-terms are an appropriate means for representing *vblock* ::= *variable* | ”(” { *vblock* ”,” } *vblock* ”)”

expr ::= *variable* | *constant* | ”(” { *expr* ”,” } *expr* ”)” |
operator ”(” *expr* ”)”

dfg-term ::= ”λ” *vblock* ”.” {”let” *vblock* ”=” *expr* ”in”} *expr*

sequential circuit ::= ”automaton” ”(” *dfg-term* ”,” *expr* ”)”

block ::= ”PARTIALIZE” *basic_block* | ”WHILE” *condition block* |
block ”THEN” *block* | ”IFTE” *condition block block* |
”LOCVAR” *constant block* |
”LEFTVAR” *block* | ”RIGHTVAR” *block*

program ::= ”PROGRAM” *constant block*

algorithmic dfg-circuit description ::=
dfg-interface pattern ”(” *dfg-term* ”,” *number of cycles* ”)”

algorithmic p-circuit description ::=
p-interface pattern ”(” *program* ”)”

s-interface ::= ”(” ”reset” { ”,” *channel* } ”)”

s-process ::= *algorithmic dfg-circuit description* |
algorithmic p-circuit description |
k-process

s-structure ::= ”∃” *channel* { ”,” *channel* } ”.”
s-process *s-interface* { ”∧” *s-process* *s-interface* }

Figure 1: Syntax of Gropius

boolean operators		AND, OR, INV, T, F
abstract operators	- *polymorphism*	MUX, EQ
	- *enumeration types*	enum, next
	- *arrays*	mkarray, spread, pick, modify, cut, append, shift, rev, comb, split, shrink, unshrink, ripple
	- *one element type*	one
	- *optional values*	none, any, CASE_option
	- *variant records*	INL, INR, CASE_sum
dfg-interface patterns		dfg_interface_cycle, dfg_interface_start, dfg_interface_reset, dfg_interface_pipeline, dfg_interface_system
p-interface pattern		p_interface_cycle, p_interface_start, p_interface_reset, p_interface_system
k-processes		Synchronize, Delay, Double, Join, Split, Collect, Distribute, Mark, Counter

Figure 2: Basic Constructs of Gropius

combinatorial circuit descriptions at the gate level and at the rt-level, respectively. The operator automaton is used for representing sequential circuits. Given some dfg term f representing the combinatorial part of the sequential circuit and some initial state q, automaton maps (f, q) to a sequential circuit automaton(f,q). automaton is used both for gate level circuits and rt-level circuits (see [Eise97b]).

Dfg-terms are not only used as a means for representing combinatorial parts of circuits, but are also used for describing i/o relations at the algorithmic level. At the algorithmic level the circuit designer may use arbitrary computable functions as starting point. In Gropius, however, p-terms and dfg-terms are strictly divided. For specific synthesis tasks, it is worthwhile knowing, that the function to be considered is a pure data flow graph and — other than with p-terms — nontermination is not a matter that has to be considered.

At the algorithmic level (Gropius-2), a fixed set of interface patterns is used to describe how the circuit implements some function g. They differ in the way the circuit communicates with the environment. There are two classes of interface patterns: ones related to dfg-terms and others related to p-terms.

At the algorithmic level, only single processes are considered. Gropius also supports multi-process systems at the system level (Gropius-3). In Gropius-3, systems are represented by structures of processes. In Gropius-3 there are two kinds of processes: algorithmic circuit descriptions and k-processes. Processes in Gropius-3 use a fixed communication scheme: higher order petri nets [Jens92], i.e. petri nets with marks having values. Therefore only two of the above

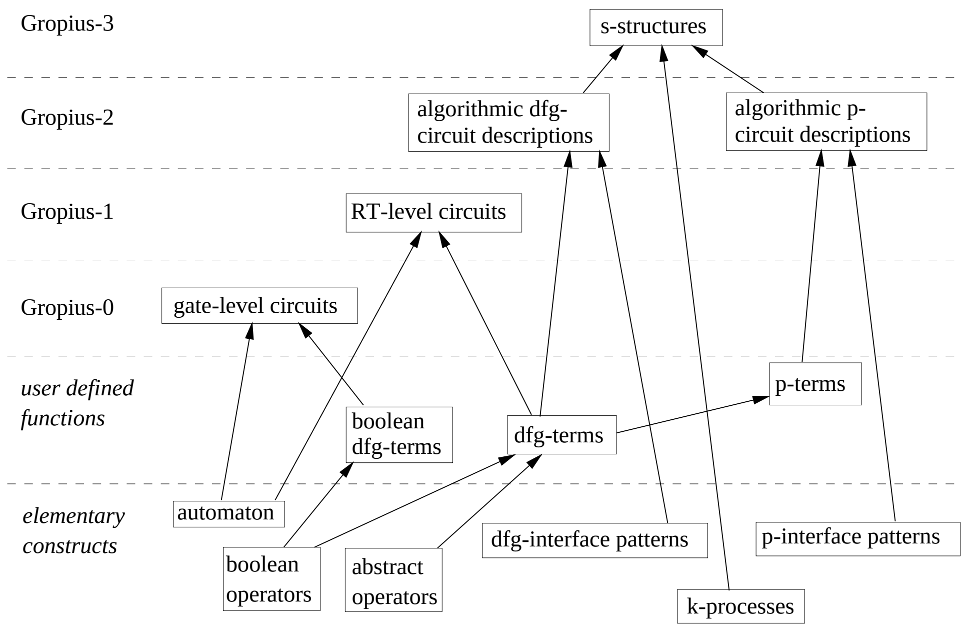


Figure 3: The language Gropius

mentioned interface patterns are allowed in Gropius-3: dfg_interface_system and p_interface_system. The k-processes are used for communication purposes: delaying marks, duplicating marks, synchronizing marks etc.. K-processes can be used for combining several processes. But it also makes sence to combine k-processes with a single interface patterns in order to modify their interface behavior. This is how the user can define new interface patterns.

In Gropius, synthesis means translating circuit descriptions from Gropius-3, Gropius-2 and Gropius-1 down to Gropius-0. In Gropius, *all* circuit descriptions are synthesizable, i.e. can be mapped to an equivalent Gropius-0 representation. Annotation: for generic circuit description a concrete instantiation has to be made and for abstract data an encoding is required.

4 Everything can be abbreviated

Gropius is pretty easy to use. The user may build an arbitrary new expression and give the expression a new name. So the user can for example define new functions:

$$\mathsf{nor}(a,b) \ := \ \mathsf{not}(\mathsf{or}(a,b))$$

Such a definition means an extension to Gropius. From now on nor may be used as an additional function. The semantics is obvious: $\mathsf{nor}(a,b)$ is nothing but an abbreviation for $\mathsf{not}(\mathsf{or}(a,b))$.

Besides functions, the user can, for example, also define new control structures. Gropius provides only a minimum number of control structures. Although there is only a WHILE-loop and the REPEAT loop is not provided, the REPEAT loop can easily be defined by the user:

$$\begin{aligned}&\mathsf{NOP} \;:=\; \mathsf{PARTIALIZE}(\lambda x.\, x)\\&\mathsf{LOOP}\ A\ c\ B \;:=\; A\ \mathsf{THEN}\ (\mathsf{WHILE}\ c\ (B\ \mathsf{THEN}\ A))\\&\mathsf{REPEAT}\ A\ c \;:=\; \mathsf{LOOP}\ A\ c\ \mathsf{NOP}\end{aligned}$$

It has to be noted, that all free variables on the right hand side of a definition must be parameters of the newly defined construct. All operators on the right hand side must already have been defined. The equation must be non-recursive, i.e. the function being defined must not appear on the right hand side.

5 Polymorphism

Gropius supports polymorphism. There are two basic polymorphic functions: a multiplexer MUX with one control bit and a variable signal type, and an equivalence gate EQ whose two input values are of some variable type. Based on these two elementary polymorphic functions, the user may derive more complex polymorphic functions such as:

$$\mathsf{MUX4}(r,s,a,b,c,d) \;:=\; \mathsf{MUX}(r, \mathsf{MUX}(s,a,b), \mathsf{MUX}(s,c,d))$$

Polymorphic functions can be used in different instantiations. The function MUX, for example, has type $\mathsf{bool} \times \alpha \times \alpha \to \alpha$, where α is a type variable. α can be instantiated in an arbitrary manner. This need not be done explicitely, but is derived from the context in which MUX is used. In the expression $\mathsf{MUX}(a, \mathsf{T}, b)$, for example, α becomes bool, whereas in $\mathsf{MUX}(a, (\mathsf{T}, b), (c, \mathsf{OR}(d,e))$ the type variable α is instantiated with $\mathsf{bool} \times \mathsf{bool}$.

6 Parameterization with Circuits

In Gropius, it is also allowed to have circuits as parameters of circuits. The following, user defined function quad maps some gate f to a structure consisting of four f gates.

$$\mathsf{quad}\ f\ (a,b) \;:=\; (f(f(a)), f(f(b)))$$

In this example f is polymorphic. Its type is $\alpha \to \alpha$. The construct quad is a very general pattern representing a simple structure of four equal combinatorial circuits in a very general manner: f may be any function of type $\alpha \to \alpha$ with α being an arbitrary type.

7 Regularity

Gropius supports regularity in a systematic manner. There is one basic regular structure named ripple. ripple is a generic, polymorphic structure of a sequence of circuits (see figure 4). Regular circuit structures lead to regular signal bundling. In Gropius, arrays are used for representing regular signal patterns. Besides ripple, Gropius provides a set of functions on arrays (see table 2). Figure 4 gives an impression on how powerful complex structures can be described in Gropius.

The definitions are build bottom up from the Gropius basics. Some of the corresponding structures are sketched in the upper part of figure 4. As can be seen, the user can start defining some general, polymorphic patterns such as ripplec and rect and later on instantiate them to derive more concrete circuits such as ANDN. One just has to instantiate the parameter n to achieve a concrete, implementable circuit. (ANDN 7) for example is an and-gate with 7 inputs.

8 Strict separation between functional and temporal aspects

At the algorithmic level (Gropius-2) as well as on the system level (Gropius-3), functional i/o description and temporal embedding are strictly separated. This means that you can easily switch from one temporal abstraction to another by just replacing the interface pattern. Other approaches use hardware representations where functional aspects and timing aspects are interwoven (VHDL etc.). This reduces the flexibility of the design process. Changing the interface behaviour means skipping the original design and starting from scratch.

Other than in many other concepts, there is a common basis for the algorithmic and for the rt- and gate level: dfg-terms. The expression (ANDN 7) can be considered as a combinatorial circuit at the rt-level. But it can as well be considered as an algorithmic description: dfg_interface_cycle(ANDN 7, 5) uniquely describes a sequential circuit implementing the n implementation with a delay (input to output) of 5 clock cycles. To derive the rt-implementation from this expression, high-level synthesis has to be applied.

Besides the nonrecursive dfg-terms, Gropius also supports p-terms, i.e. general computable functions (programs), as starting point for high level synthesis. Since dfg-terms are a basic part of p-terms, the function (ANDN 7) could as well be used as a part of some p-term — for example in a boolean condition of a while loop.

9 Conclusion

We have presented a new functional hardware description language, and we showed some of its major concepts. Due to lack of space, most concepts could only be introduced very briefly. Gropius is a formal, mathematical hardware description language with an exact semantics. It allows describing circuits at an abstract, compact, mathematical manner. Its expressiveness goes beyond the standard of nowadays hardware description languages.

References

[Eise97b] D. Eisenbiegler. Automata — A theory dedicated towards formal circuit synthesis. Technical Report 14/97, Universität Karlsruhe, 1997. http: //goethe. ira. uka.de/fsynth/publications/postscript/Eise97b.ps.gz.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[Jens92] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Springer, 1992.

[JoBo91] S.D. Johnson and B. Bose. DDD: A system for mechanized digital design derivation. In *International Workshop on Formal Methods in VLSI Design*, Miami, Florida, January 1991. ACM/SIGDA. Available as Indiana University Computer Science Department Technical Report No. 323 (rev. 1997).

[KBES96] R. Kumar , C. Blumenröhr, D. Eisenbiegler, and D. Schmid . Formal synthesis in circuit design-A classification and survey. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design. First International Conference,FMCAD'96*, number 1166 in Lecture Notes in Computer Science, pages 294–309, Palo Alto, CA, USA, November 1996. Springer-Verlag.

[KlBr95] C.D. Kloos and P.T. Breuer, editors. *Formal Semantics for VHDL*, volume 307 of *The Kluwer international series in engineering and computer science.* Kluwer, Madrid, Spain, March 1995.

[LLLA93] J.O. Leary, M. Linderman, M. Leeser and M. Aagaard. HML: A hardware description language based on standard ML. In D. Agnew, L. Claesen, and R. Camposano, editors, *IFIP Conference on Hardware Description Languages and their Applications*, volume A-32, pages 327–334, Ottawa, Ontario, Canada, April 1993. IFIP WG10.2 International Conference, North-Holland.

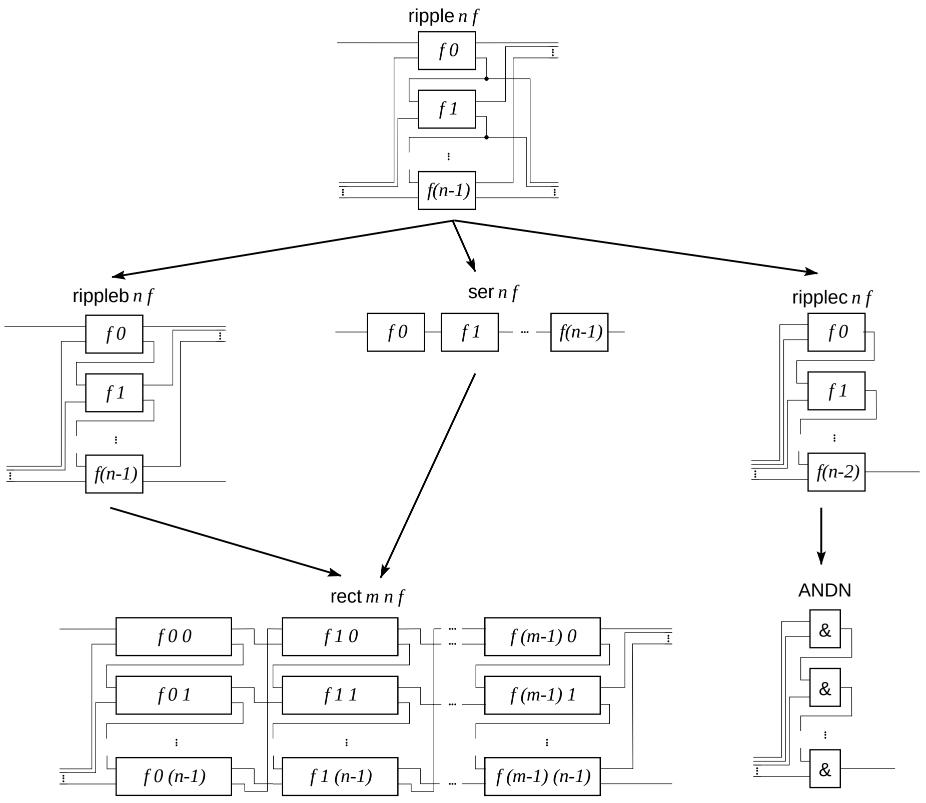
[MoCl93] J.D. Morison and A.S. Clarke. *ELLA 2000: A language for electronic system design.* McGraw-Hill, 1993.

[RaPN97] M. Radetzki, W. Putzke-Rming, and W. Nebel. Objective VHDL: The Object-Oriented Approach to Modeling and Design. In *EMMSEC'97 proceedings.* 1997.

[ShRa95] R. Sharp and O. Rasmussen. The T-Ruby design system. In *IFIP Conference on Hardware Description Languages and their Applications*, pages 587–596, 1995.

[SwMC95] S. Swamy, A. Molin, and B. Covnot. OO-VHDL: OO-VHDL Object-Oriented Extensions to VHDL. In *transactions on computer science*, pages 18–26. IEEE, 1995.

[VHDL96] IEEE. *IEEE Standard VHDL Language Reference Manual Std 1076.3*, 1996.



$\mathsf{EQP}\ m\ n$	$:=$	$\mathsf{EQ}(\mathsf{enum}\ (m+n+1)\ m, \mathsf{enum}\ (m+n+1)\ n)$
$\mathsf{first}\ n\ x$	$:=$	$\mathsf{pick}\ n(\mathsf{enum}\ n\ 0, x)$
$\mathsf{maxenum}\ n$	$:=$	$\mathsf{enum}\ n\ (n-1)$
$\mathsf{last}\ n\ x$	$:=$	$\mathsf{pick}\ n(\mathsf{maxenum}\ n, x)$
$\mathsf{last'}\ n\ (x,y)$	$:=$	$\mathsf{MUX}(\mathsf{EQP}\ n\ 0, y, \mathsf{last}\ n\ x)$
$\mathsf{constantly}\ f\ i$	$:=$	f
$\mathsf{ser}\ n\ f\ x$	$:=$	$\mathsf{last'}\ n\ ($ $\mathsf{SND}(\mathsf{ripple}\ n\ (\lambda i.\ \lambda(a,b).\ (\mathsf{one}, f\ i\ a))\ (x, \mathsf{spread}\ n\ \mathsf{one})),$ $x\ )$
$\mathsf{rippleb}\ n\ f\ (a,b)$	$:=$	$\mathsf{let}\ (c,d) = \mathsf{ripple}\ n\ f\ (a,b)\ \mathsf{in}\ (c, \mathsf{last'}\ n\ (d,a))$
$\mathsf{ripplec}\ n\ f\ (a,b)$	$:=$	$\mathsf{last'}\ n\ (\mathsf{SND}(\mathsf{ripple}\ n\ (\lambda i.\ \lambda x.\ (\mathsf{one}, f\ i\ x)))\ (a,b),\ a)$
$\mathsf{rippled}\ n\ f\ (a,b)$	$:=$	$\mathsf{MUX}(\mathsf{EQP}\ n\ 0,\ d,$ $\mathsf{MUX}(\mathsf{EQP}\ n\ 1,\ \mathsf{first}\ n\ x,$ $\mathsf{ripplec}\ (n-1)\ f(\mathsf{first}\ n\ x, \mathsf{shift}\ 1\ (n-1)\ x)\ )\ )$
$\mathsf{rect}\ m\ n\ f\ (a,b)$	$:=$	$\mathsf{ser}\ m\ (\lambda i.\ \lambda(x,y).\ \mathsf{swap}(\mathsf{rippleb}\ n\ (f\ i)\ (x,y))\ (a,b))$
$\mathsf{ANDN}\ n\ x$	$:=$	$\mathsf{rippled}\ n\ (\mathsf{constantly}\ \mathsf{AND})\ (\mathsf{T}, x)$

Figure 4: Examples of derived structures