https://doi.org/10.20378/irbo-52666

Constrained Proofs: A Logic for Dealing with Behavioural Constraints in Formal Hardware Verification[*]

Michael Mendler[†]
Department of Computer Science
University of Edinburgh

Abstract

The application of formal methods to the design of correct computer hardware depends crucially on the use of abstraction mechanisms to partition the synthesis and verification task into tractable pieces. Unfortunately however, behavioural abstractions are genuine mathematical abstractions only up to behavioural *constraints*, i.e. under certain restrictions imposed on the device's environment. Timing constraints on input signals form an important class of such restrictions. Hardware components that behave properly only under such constraints satisfy their abstract specifications only approximately.

This is an impediment to the naive approach to formal verification since the question of how to apply a theorem prover when one only knows *approximately* what formula to prove has not as yet been dealt with.

In this paper we propose a notion of *constrained proof* and *constrained proposition* which provides for 'approximate' verification of abstract specifications and yet does not compromise the rigour of the argument. It is based on the idea of removing the constraints from the specification and making them part of its proof. Thereby the abstract verification is separated from constraint analysis which in turn may be delayed arbitrarily. We have implemented the logic on the interactive theorem prover LEGO and verified simple examples. The presentation in this paper uses one of these examples for explaining the problem and demonstrating the use of the logic.

[*]This work was supported by the *Studienstiftung des deutschen Volkes* and SERC grant No.GR/F 35890 "Formal System Design"

[†]Now at University of Erlangen-Nürnberg, Institute for Computer Aided Circuit Design, email *mendler@faui77.informatik.uni-erlangen.de* or *mm@lfcs.ed.ac.uk*

1 Introduction

Quite a lot of work has been done to make hardware verification practicable at a non-academic scale [5,4,19,20,25]. What still seriously limits its success is the almost insurmountable complexity from which verification of non-trivial hardware designs suffers. First steps are being undertaken by various researchers to exploit structuring concepts such as *modularisation* and *abstraction* in the design of hardware to break up the verification task into a series of smaller chores, each of which can be dealt with independently [5,4,26,18,35,32,10]. The question arises how this approach should best be implemented in modern interactive theorem provers.

1.1 The Problem of Constraints

A typical phenomenon one encounters with the implementation of even conceptually simple abstraction steps which are standard practice in hardware engineering is that they cannot be formalised without introducing *constraints.* Constraints are assumptions about the device's environment under which the particular abstraction (of its behaviour) at hand is actually valid.

An example is the passage from a sequential circuit, built according to the synchronous design paradigm, to its abstract description in terms of an input-output automaton. Here the abstraction is only valid as long as the environment (among other things) obeys a *timing constraint* which says that all input lines of the sequential circuit must be kept stable during a certain well-defined phase of the clock. Clearly, the necessity for imposing timing constraints is a general phenomenon, not restricted to the synchronous case. It is an even more important issue in asynchronous designs. [17,34,31].

The interaction between abstraction and constraints poses a tangled problem. Constraints interfere with the essential idea of reasoning about a behaviour in *abstract* terms which is to avoid details specific to the implementation at the more *concrete* level. For it is impossible to work with the device's abstract behaviour without at the same time having to deal with the concrete-level constraints on which it depends. To verify, for instance, that the behaviour of a composite device meets its abstract specification it does not suffice simply to compose the abstract specifications of its components. The verification also has to show that at the concrete level the composition does not violate the constraints of each component. This, in general, will make it necessary to impose constraints again on the environment of the composite device.

Thus, constraints defeat the idea of top-down refinement, which is first to decompose a system into components at the abstract level and then independently to implement each component at the concrete level; Verifying constraints requires knowledge both of the overall structure of the system (the environment of a component) from the abstract level *and* of the implementation (the constraints of a component) at the lower level. In short, the general situation in the modeling of hardware seems to be that of incomplete abstractions, i.e. abstractions *modulo constraints.* The constraints on which the abstraction depends embody residual aspects of the concrete level that impinge on the subsequent design and cannot be abstracted away once and forall.

1.2 A Possible Solution

The best one can expect here is to find means by which *reasoning about abstract behaviour* and *constraint analysis* fall into two separate verification passes rather than having them intertwined as the straight-forward approach suggests. The goal of this paper is to introduce and justify a logic in which the main verification of an abstract behaviour is truly an abstract verification in that it does not have to be concerned with constraints. It proceeds by assuming a successful constraint analysis wherever it depends on constraints. In the course of this main verification information about the constraints is accumulated as a proof obligation to be filled in at a later stage. Ideally, the remaining verification task corresponding to constraint analysis would then be handed over to a specialised tool. In some cases it could be done automatically, for instance extracting the minimal clock period for a synchronous system. In other cases, where the logic is undecidable, it has to be done manually. An example of this would be proving that the output of a certain integer function lies within a given finite range.

The idea leading to the proposed logic presented in this paper is not new. It reflects good engineering practice: In a first approximation one tries to establish the feasability of a design. Only then is it worthwhile to attempt a complete validation in a second step. New however, is the attempt to formalise this engineering principle mathematically and to implement it at the root of a theorem prover.

We have implemented an experimental prototype of the logic with the help of the mechanical proof checker LEGO [29]. LEGO provides a bed for encoding natural deduction style logics in the type theory of the Logical Framework (LF) [16] or the Calculus of Constructions (CC) [6,7] plus some extensions. We have used an extension of CC [22] which, as the most prominent feature, provides Σ-types, i.e. a generalisation of ordinary pairing where the *type* of the second component in a pair may depend on the *value* of the first component (ordinary pairing is the special case in which the types of both components are independent from their respective values). The following will show that it is in fact the Σ-types that make it possible to cut out and delay constraint analysis, so that the main proof can be performed without looking at or even manipulating constraints. It merely records necessary information about what has to be proved later, when one turns to analysing the constraints.

It is outside the scope of this paper to give an introduction to LEGO and to present the implementation in detail. Instead, we will use an informal mathematical language which, as we hope, will serve to explain the underlying idea in a compact and lucid way. We would like to stress that everything presented in this paper actually can be done interactively, modulo syntax, with the LEGO system even where we do not mention it explicitly.

2 Constraints and Synchronous Hardware Design

We are going to explain the implemented logic by means of a simple example that is just complex enough to convey the basic idea. As the area of hardware design where the example is taken from we have chosen the particular form of temporal abstraction that is fundamental to the design of synchronous hardware. Let us first briefly explain the general situation (Section 2.1) and then turn to a concrete example (Section 2.2).

2.1 Synchronous Circuits

A typical synchronous circuit is built up from *latches* (such as D-type flipflops) and *combinational circuits* (such as nand gates, inverters, and nets thereof). In a slightly simplified view[1] one can summarise the essence of the synchronous design paradigm in the following *design rules*:

C1 All latches are triggered by a common clock signal

C2 There is at least one latch in every feedback loop

C3 The clock period is long enough to allow for signal changes caused by any clock event to settle throughout the circuit before the next clock event

C4 The inputs to the circuit have to be stable long enough prior to any clock event for signals to have become stable by the clock event.

In a broad sense all of these design rules can be interpreted as constraints, more precisely, C1-C2 as *structural* constraints and C3-C4 as *behavioural* constraints. From a verification point of view the structural constraints C1-C2 are essentially reflections of internal behavioural constraints, i.e. they are conditions necessary for verifying that no behavioural constraints are violated by components within the circuit.

Much of the success of the synchronous design style is due to the fact that under the design rules C1-C4 one does not need to consider propagation delays when reasoning about the circuit's behaviour. If one is interested in the state of the circuit only at every clock event (or during a certain interval around it) and records the evolution of input and output values at these points, then the descriptive effort can be drastically reduced:

A1 Latches behave like unit delays

A2 Combinatorial circuits behave like delay free boolean functions

A3 The complete synchronous circuit reduces to a finite automaton and the automaton's behaviour can be derived by composing unit delays and delay-free boolean functions. More precisely, every unit delay gives rise to one state variable and the state transition function is determined by the interconnection of state variables through boolean functions.

Thus, relativising the synchronous circuit's behaviour to the *abstract time* given by the succession of clock events abstracts from propagation delays. Note, that the restriction on clock events can also be viewed as part of the design rules and as a constraint on the usage of the circuit which is characteristic to synchronous abstraction.

Although, either implicitly or explicitly, *timing abstraction* has always been used in the design of synchronous systems [36,3,8,23], it seems that first attempts to formalise it for the purpose of verification have only recently been made [26,18]. The separation of design rule checking (C1–C4) from reasoning in abstract terms (A1–A3) is crucial for practical applications, but there seems to be no satisfactory implementation of this separation on an interactive theorem prover. For instance, Herbert's methodology [18], implemented on

[1]We ignore here, among other things, for the sake of simplicity set-up and hold times of latches or the possibility to use multiple clocks. This does not, however, affect the point.

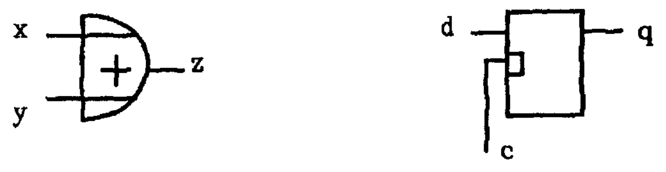

Figure 1: xor-gate and level triggered latch

the proof checker HOL [11,12], though it conceptually distinguishes between statements about timing and abstract behaviour, leaves both aspects intertwined at the level of proofs. This basically means that design rule checking and reasoning in abstract terms have to go together in a single proof. The logic presented in this paper provides a way to separate these concerns within a single logical inference system.

2.2 A Simple Circuit Design

Let us turn to a specific example. Take the simple case of a combinational circuit such as a 'xor' gate and a level triggered latch (Figure 1) which, as in [18], are to be considered as components of a synchronous system; i.e. they are put into an environment with a global clock, relative to which certain conditions on the stability of inputs can be imposed as *timing constraints* to allow timing abstraction.

Their behaviour can be described by predicates over input and output signals. For simplicity we take signals to be functions from integers to booleans, i.e.

$$signal = int \rightarrow bool$$

Assuming that both gates have constant propagation delays $\delta_{xor} > 0$ and $\delta_{latch} > 0$, their behaviour may be defined by the following axioms:

$$\begin{aligned}
xor\,(x,y,z) &\overset{df}{=} \forall t : int.\ z(t + \delta_{xor}) = xt + yt \\
latch\,(d,c,q) &\overset{df}{=} \forall t : int.\ (ct = 1 \supset q(t + \delta_{latch}) = dt)\ \wedge \\
&\qquad (ct = 0 \supset q(t + \delta_{latch}) = q(t + \delta_{latch} - 1))
\end{aligned}$$

Note, that we are using the operator $+$ both for addition over *int* as well as for modulo-2 sum over *bool*. According to the axioms the xor-gate performs the modulo-2 sum of its inputs x, y at every time step and outputs them with a delay δ_{xor} on output z. The latch is enabled to pass data from input d to output q with a delay δ_{latch} by positive levels of the clock input c, and it is locked when $ct = 0$.

For the purpose of this paper these simple axioms are assumed to be the low-level, most detailed, description available for the components *xor* and *latch*. Clearly, they are already an abstraction of real devices' behaviours. A more realistic description would have to account for variable gate delays as well as setup and hold times for the latch; it would perhaps assume continuous rather than discrete time and signal values, require maximal signal rise and fall times, and so on. Since for describing our logic it is of no importance how detailed a model of behaviour one actually starts from we have taken the simplest axioms possible.

The reader is referred to [2,14] for a discussion of more sophisticated axiomatizations of elementary digital circuits.

The important thing to note is that *xor* and *latch* contain both timing (delays) and functional aspects (operations on booleans) intertwined. In a synchronous design context, however, where one takes advantage of the design rules, one expects not having to care about delays. More precisely, the xor-gate should behave like a delay-free boolean function and the latch like an one-unit delay (A1-A2). Therefore, in place of *xor* and *latch* one would rather work with axioms like

$$\begin{aligned} \mathit{xor\_syn}\,(x, y, z) &\overset{df}{=} \forall t : \mathit{int}.\ zt = xt + yt && (1)\\ \mathit{latch\_syn}\,(d, q) &\overset{df}{=} \forall t_1, t_2 : \mathit{int}.\ \mathit{next}\,(t_1\,,\,t_2) \supset q(t_2) = d(t_1) && (2) \end{aligned}$$

where *next* $(t_1\,,\,t_2)$ is a predicate expressing that t_1 and t_2 are two consecutive points in time. It is defined abstractly[2] by

$$\mathit{next}\,(t_1\,,\,t_2) \overset{df}{=} t_1 < t_2 \ \land\ \forall t : \mathit{int}.\, t_1 < t \supset t_2 \le t$$

In this abstract view the clock does no longer appear as an input to the latch. For in a synchronous circuit the latch's clock input is always connected to the global clock signal and consequently no longer available as an input. Thus, assume a clock signal

$$\mathit{clk} : \mathit{signal}$$

which is globally defined throughout the system. As a result *clk* may be used within formulae without it being mentioned explicitly as a parameter.

Obviously, *xor_syn*, *latch_syn* cannot be proved from *xor* and *latch* right away since the delays cannot be wiped out. What can be proved, however, by introducing constraints are certain approximations thereof. Before we can state them we need some predicates for formulating constraints. We first assume that clock ticks are marked by rising edges of *clk* and define a corresponding predicate

$$\mathit{tick}\, t \overset{df}{=} \mathit{clk}(t-1) = 0 \ \land\ \mathit{clk}(t) = 1$$

which obtains true if there is a clock tick at time t. Given this predicate one may define what it means that a signal x is stable in all intervals of length δ prior to clock events:

$$\mathit{stable}\, x\, \delta \ \overset{df}{=}\ \forall t_1, t_2 : \mathit{int}.\ (\mathit{tick}\, t_1 \ \land\ t_1 - \delta \le t_2 \le t_1) \supset x(t_1) = x(t_2)$$

Finally, for constraining the clock we have two other predicates, the first expressing that the 1-phase of the clock lasts exactly one time step, and the second imposing a minimal distance δ on two consecutive clock ticks:

$$\begin{aligned} \mathit{one\_shot} &\overset{df}{=} \forall t : \mathit{int}.\ \mathit{clk}(t) = 1 \supset (\mathit{clk}(t-1) = 0 \land \mathit{clk}(t+1) = 0)\\ \mathit{min\_sep}\,\delta &\overset{df}{=} \forall t_1, t_2 : \mathit{int}.\ (t_1 < t_2 \ \land\ \mathit{tick}\, t_1 \land \mathit{tick}\, t_2) \supset t_2 \ge t_1 + \delta \end{aligned}$$

[2]One might want to turn the predicate *next* into a function which for time t yields the successive time step *next*(t) and is undefined otherwise. In a logic with partial terms this could be done using the so-called ι-operator which we do not have available in LEGO.

With these predicates put into place the promised approximations of *xor_syn* and *latch_syn* can be formulated:

$$\begin{aligned} \mathit{xor\_abs}\,(x,y,z) &\overset{df}{=} \;\; \textbf{\textit{stable}}\; \boldsymbol{x\,\delta_{xor}} \;\wedge\; \textbf{\textit{stable}}\; \boldsymbol{y\,\delta_{xor}} \\ &\quad \supset \forall t : \mathit{int}.\; \textbf{\textit{tick}}\; \boldsymbol{t} \;\supset\; zt = xt + yt \\ \mathit{latch\_abs}\,(d,q) &\overset{df}{=} \;\; (\textbf{\textit{one\_shot}} \;\wedge\; \textbf{\textit{min\_sep}}\; \boldsymbol{\delta_{latch}}) \supset \forall t_1, t_2 : \mathit{int}.(\textbf{\textit{tick}}\; \boldsymbol{t_1} \;\wedge\; \textbf{\textit{tick}}\; \boldsymbol{t_2} \\ &\quad \supset (\mathit{next\_abs}(t_1\,,\, t_2) \;\supset\; q(t_2) = d(t_1))) \end{aligned}$$

where *next_abs* is the following approximation of *next*:

$$\mathit{next\_abs}(t_1\,, t_2) \overset{df}{=} t_1 < t_2 \;\wedge\; \forall t : \mathit{int}.\; \textbf{\textit{tick}}\,t \;\supset\; (t_1 < t \;\supset\; t_2 \leq t)$$

The bold-faced parts indicate the offset of the approximations from the ideal versions *xor_syn* and *latch_syn*. This offset explicitly reflects the design rules (C3-C4): timing constraints on inputs, on the clock signal, and the sampling at clock events. In contrast to *xor_syn*, *latch_syn* these approximations now can be derived from axioms *xor* and *latch*, i.e. we have

$$\begin{aligned} \mathit{xor}\,(x,y,z) &\;\vdash\; \mathit{xor\_abs}\,(x,y,z) \\ \mathit{latch}\,(d,\mathit{clk},q) &\;\vdash\; \mathit{latch\_abs}\,(d,q) \end{aligned}$$

We state this without proof here. The gap will be filled later when we have introduced the logic. Note, that due to the simplification of the latch's behaviour (i.e. no set-up and hold conditions) *latch_abs* does not require stability of data input *d* relative to the clock.

The observation that stability constraints essentially work to squeeze out delays of the behavioural description and thereby separate timing behaviour from functional behaviour is already employed in [18,26]). Here we push this idea further so as to also encompass constraints on time points, i.e. *tick t*. Being restrictions which also reflect the design rules, constraints on time points should be subjected to the same treatment as are stability constraints on signals. In fact, our logic will also deal with this type of constraints.

Now suppose as a simple design task, we wanted to build a stoppable *modulo*-2 counter. It is to have one input and one output, and to produce a stream of alternating 0s and 1s as long as the input is at 1 and stop at the current output value when the input switches to 0. More formally, its behaviour is specified by the following logical formula:

$$\mathit{cnt\_spec}\,(x,y) \overset{df}{=} \forall t_1,t_2 : \mathit{int}.\; \mathit{next\_abs}\,(t_1,t_2) \supset y(t_2) = x(t_2) + y(t_1)$$

From this input-output specification one derives easily a Moore automaton or equivalently an implementation consisting of an exclusive-or function and a one-unit delay as depicted in Figure 2. Given, that *xor_syn* (1) describes the behaviour of the exclusive-or and *latch_syn* (2) that of the one-unit delay, the behaviour of the implementation is given by

$$\mathit{cnt\_syn}\,(x,y,z) \overset{df}{=} \mathit{xor\_syn}\,(x,z,y) \;\wedge\; \mathit{latch\_syn}\,(y,z) \tag{3}$$

which employs logical *conjunction* $\wedge$ of predicates to express *composition* or *superposition* of two behaviours. Another important operation on behaviours is *hiding* of internal wires which logically is achieved by *existential quantification*. Since in the example the specification of the counter describes a circuit with one input and one output we have to consider signal z as internal in the implementation (3), i.e. x and y are the required input and output

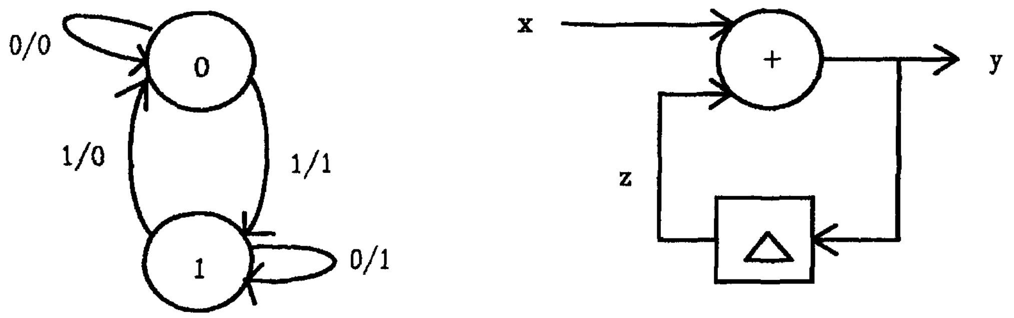

Figure 2: Implementation of the *modulo*-2 counter

signals visible to the environment. Verifying that the implementation is correct would now amount to proving that after hiding of the internal signal z the implementation (3) entails the specification, i.e.

$$\exists z : \mathit{signal}.\ \mathit{cnt\_syn}\,(x,y,z) \ \vdash\ \mathit{cnt\_spec}\,(x,y) \qquad (4)$$

This would be an easy exercise invoking the rules of ordinary first-order logic. Unfortunately, applying synchronous abstraction to *xor* and *latch* does not provide an ideal exclusive-or or an ideal one-unit delay satisfying *xor_syn* and *latch_syn* but merely approximations *xor_abs* and *latch_abs*. Therefore the implementation we are actually able to get is

$$\mathit{cnt\_abs}\,(x,y,z) \stackrel{df}{\equiv} \mathit{xor\_abs}\,(x,z,y)\ \wedge\ \mathit{latch\_abs}\,(y,z)$$

Of course there is no reason to expect that $\exists z : \mathit{signal}.\ \mathit{cnt\_abs}\,(x,y,z)$ entails $\mathit{cnt\_spec}\,(x,y)$. Rather, in place of the original *cnt_spec*, we will again only achieve an approximation, perhaps something of the form

$$\begin{aligned}\mathit{cnt\_appr}\,(x,y) \ \stackrel{df}{\equiv}\ & (C_0\ \wedge\ C_1(x,y))\ \supset\ (\forall t_1,t_2 : \mathit{int}.\ C_2(t_1,t_2)\\ & \supset\ (\mathit{next\_abs}\,(t_1,t_2)\ \supset\ y(t_2) = x(t_2) + y(t_1)))\end{aligned}$$

where C_0, C_1, and C_2 are constraints that have to be imposed on the composite circuit to allow the envisaged derivation

$$\exists z : \mathit{signal}.\ \mathit{cnt\_abs}\,(x,y,z) \ \vdash\ \mathit{cnt\_appr}\,(x,y) \qquad (5)$$

Here we are facing the question of how to go about finding the constraints C_0, C_1, C_2 and thus the modified specification *cnt_appr*. The straightforward approach, as employed for instance in [14], is attempting a derivation of *cnt_spec* (from *cnt_abs*), finding out where it fails, and at each such dead end identifying assumptions that would make it work if they were available in the first place. This information can then be used for determining the constraints C_0, C_1, C_2 and the place where they have to go to weaken the specification appropriately. This is however not quite satisfactory since it means going through the verification proof twice, once for finding the constraints and a second time after pasting them into the specification for completing the proof. Furthermore, and more importantly, the proof has to intermingle timing constraints with abstracted properties; it aims to prove the abstract specification *cnt_spec* while at the same time having to deal with the constraints inside the propositions *xor_abs*, *latch_abs*, *'next_abs*, and *cnt_appr*.

As argued before, this is not what one really would like to do. Rather, one would like first to perform the abstract verification (4) *without* consideration of constraints. This establishes the feasibility of the design at the abstract level. The constraints, which are dependent on a particular implementation mechanism -here the implementation as a synchronous circuit-, are not determined before the implementation of the abstract components is chosen. In the example, this leads to the approximations *xor_abs*, *latch_abs*. Finally, a constraint analysis should be able to use the abstract proof (4) together with the knowledge of the constraints contained in *xor_abs* and *latch_abs* for extracting the constraints in *cnt_appr*.

In the following we will show how to achieve this goal by reformulating the notions of proof and proposition so as to 'hide' constraints within them and set up a rudimentary calculus of derivations to deal with this *constrained* logic.

3 Logic of Constrained Proofs

The logic will consist of a suitable base logic in which to express both the constraints and the abstraction properties of the verification example. For the purpose of our particular example it is sufficient to assume a formal system for typed first order logic. It is not really essential what the base logic looks like as long as it fulfills certain minimal requirements which we set out below. This base logic will then be extended by a new syntactic operator [.] for 'hiding' constraints. The described logic of constrained proofs therefore need not be seen as a particular and fixed logic but rather as a method for extending ones favourite logic for accomodating constraints.

In the following we assume some familiarity with natural deduction style logics and the lambda calculus (with explicit pairing) as a simple functional programming language.

3.1 Base Logic

Of the base logic we will require logical connectives for conjunction $\phi\wedge\psi$, implication $\phi\supset\psi$, universal and existential quantification $\forall x : A.\phi$, $\exists x : A.\phi$ together with the usual natural deduction rules

$$\wedge I : \frac{\phi\wedge\psi}{\phi \quad \psi} \qquad \wedge E_l : \frac{\phi}{\phi\wedge\psi} \qquad \wedge E_r : \frac{\psi}{\phi\wedge\psi} \qquad \supset E : \frac{\psi}{\phi\supset\psi \quad \phi} \qquad \supset I : \begin{array}{c} \phi\supset\psi \\ \hline \psi \\ \vdots \\ \underline{\phi} \\ \surd \end{array}$$

$$\forall E_t : \frac{\phi\{t/x\}}{\forall x : A.\phi} \qquad \forall I_x : \frac{\forall x : A.\phi}{\phi} \qquad \exists I_t : \frac{\exists x : A.\phi}{\phi\{t/x\}} \qquad \exists E_x : \begin{array}{cc} \multicolumn{2}{c}{\psi} \\ \hline \exists x : A.\phi & \psi \\ & \vdots \\ & \underline{\phi} \\ & \surd \end{array}$$

Here $\phi\{t/x\}$ denotes the substitution of term t for variable x in ϕ. Additionally, the logic comprises negation $\neg\phi$, disjunction $\phi \vee \psi$, the propositional constants $\top$ for *true* and $\bot$ for *contradiction* with rules

$$\vee I_l : \frac{\phi \vee \psi}{\phi} \qquad \vee I_r : \frac{\phi \vee \psi}{\psi} \qquad \vee E : \frac{\psi}{\begin{array}{ccc} \phi_1 \vee \phi_2 & \psi & \psi \\ & \vdots & \vdots \\ & \frac{\phi_1}{\surd} & \frac{\phi_2}{\surd} \end{array}}$$

$$\neg I : \frac{\neg\phi}{\begin{array}{c} \bot \\ \vdots \\ \frac{\phi}{\surd} \end{array}} \qquad \neg E : \frac{\bot}{\phi \quad \neg\phi} \qquad \bot E : \frac{\phi}{\bot} \qquad \top I : \frac{\top}{\phi}$$

These rules are to be understood in the usual way in what regards free and bound variables, substitution of a term for a free variable and the variable restrictions associated with rules $\forall I_x$ and $\exists E_x$. In $\forall I_x$, x must not occur in any assumption on which ϕ depends, and in $\exists E_x$, x must not occur in ψ or in any other assumption save ϕ on which the lower occurrence of ψ depends. The letter I in the name of a rule stands for 'introduction' and E for 'elimination'.

For (refinement) proofs these rules are read top-down, i.e. they reduce proving the proposition above the rule bar to proving the propositions below it. In what follows all rules are written in this top-down way. It is useful to have *syntactic definition* available in the logic as a means for abbreviating a complex formula by some user-defined name possibly with syntactic parameters. Definitions also have their introduction and elimination rules, e.g. a definition $\phi' \overset{df}{=} \phi$ is acompanied by rules

$$dfI : \frac{\phi'}{\phi} \qquad dfE : \frac{\phi}{\phi'}$$

For the logic of constrained proofs we will not only have to treat propositions but also proofs as mathematical entities which are manipulated by the inference system. All modern interactive theorem provers, like HOL, VERITAS, LAMBDA, IPE [30], ELF [13], or LEGO are based on this principle as they are essentially programming systems for manipulating proofs. A formula is identified with the set of its proofs and a rule is implemented by a program which transforms proofs of the rule's hypotheses into proofs of its conclusion. In this spirit we associate with each of the above rules a program which implements the rule bottom-up, i.e. it can be applied to proofs of the propositions below the bar to yield a proof for the proposition above it. The name of each rule will be taken to denote its associated program on proofs. Consider the rules $\wedge I$, $\wedge E_l$ and $\wedge E_r$. The introduction rule $\wedge I$ describes how to build proofs of a formula $\phi \wedge \psi$ from proofs of its components ϕ and ψ; it serves to *introduce* the junctor $\wedge$. Rules $\wedge E_l$, $\wedge E_r$ say that from a proof of $\phi \wedge \psi$ proofs of the components ϕ and ψ can be extracted; it serves to *eliminate* the junctor $\wedge$.

To be more precise, as we are dealing with rule schemata rather than single rules we would have to instantiate each rule name with the actual propositions to denote a particular program on proofs. For instance, we would write $\wedge I(\phi, \psi)$ for the function which takes pairs of proofs of ϕ and ψ to a proof of $\phi \wedge \psi$. However, since it is clear from the proofs to which $\wedge$-introduction is applied which instance of the schemata is meant we may simply write $\wedge I$. The same applies to the other rules. A nice feature of LEGO is that it supports this style of *polymorphism.* It knows to infer the type of a function from the type of the arguments to which it is applied (or, more generally, from the context in which it is used).

We do not need to know what the basic programs implementing the rules look like nor what exactly a proof is. All we want is compose them to form derived rules or *derivations* and assume that they satisfy certain natural equations guaranteeing that they interact in a coherent way. Of such equations only those are given below which are used in the sequel.

First some notation: The language chosen to compose and reason about derivations is essentially a typed lambda calculus with explicit pairing. To denote that p is a proof of ϕ we write $p : \phi$ and for f a derivation of ψ from ϕ we write $f : \phi \rightarrow \psi$.[3] So, for instance, the (basic) derivation $\wedge I(\phi, \psi) : \phi \times \psi \rightarrow \phi \wedge \psi$ may be applied to proofs $p : \phi$ and $q : \psi$ to yield a proof $\wedge I(\phi, \psi)(p, q) : \phi \wedge \psi$. As remarked above this is more simply written $\wedge I(p, q) : \phi \wedge \psi$. λ-abstraction is written as usual, and $\circ$ stands for the sequential composition of derivations in reversed order (which matches with the bottom-up notation of proof trees). The operator $\circ$ is an abbreviation definable in terms of λ-abstraction; for instance

$$\wedge E_l \circ \wedge I \;=\; \lambda x : \phi \times \psi.\ \wedge E_l(\wedge I(x)) \;: \phi \times \psi \rightarrow \phi$$

The equations required to hold between the basic rules all state that an introduction rule can be cancelled by subsequent application of the corresponding elimination rule.

$$\begin{aligned}
\wedge E_l \circ \wedge I(p : \phi\, ,\, q : \psi) &= p && (6)\\
\wedge E_r \circ \wedge I(p : \phi\, ,\, q : \psi) &= q && (7)\\
\supset E(\supset I(f : \phi \rightarrow \psi)\, ,\, p : \phi) &= f(p) && (8)\\
\exists E_x(\exists I_t(p : \phi\{t/x\})\, ,\, f : \phi \rightarrow \psi) &= f\{t/x\}(p) && (9)\\
\forall E_t \circ \forall I_x(p : \phi) &= p\{t/x\} && (10)\\
dfE \circ dfI(p : \phi) &= p && (11)
\end{aligned}$$

3.2 Extending the Base Logic

On top of this base logic we encode the idea of a *constrained proposition* proofs of which, also called *constrained proofs*, are allowed to have hidden assumptions. In the example hidden assumptions will be behavioural constraints. To introduce the general concept, we begin with the central definition.

Definition 1 *(Motivation) Let ϕ be a proposition. A* constrained proof *of ϕ is a pair $c = (\gamma, p_\gamma)$ consisting of a proposition γ and a proof of $\gamma \supset \phi$. The set of constrained proofs of ϕ will be denoted by $[\phi]$ and referred to as a* constrained proposition.

The motivation for this definition in view of the envisaged application for handling constraints is the following: Constructing a proof $c = (\gamma, p_\gamma)$ of a specification of the form $[\phi]$ amounts to proving ϕ under *some* assumption γ, for instance a timing constraint. The

[3]Note the difference between $p : \phi \supset \psi$ and $f : \phi \rightarrow \psi$; in the former p is a *proof*, in the latter f is a *derivation* (a program on proofs).

assumption is determined by the proof c and recorded as its first component. Given such a constrained proof $c : [\phi]$, one can extract both the hidden constraint $\pi_1(c)$ and the proof

$$\pi_2(c) : \pi_1(c) \supset \phi$$

that this constraint implies ϕ (π_1 and π_2 denote first and second projection, respectively). The constraint $\pi_1(c)$ may be subjected to constraint analysis and from the second component $\pi_2(c)$ one may build proof of ϕ for any proof of the constraint $\pi_1(c)$ via the $\supset E$ rule.

Definition 1 suggests to extend the base logic by a new syntactic operator $[.]$ which for any proposition ϕ forms the constrained proposition $[\phi]$. This operator as we shall see should work for all propositions, not just for those of the base logic. More precisely, we want to build propositions like

$$[\forall t : A.[\phi]],$$

i.e. we want to iterate the operator. As to the associated rules for $[\,.\,]$, the discussion above yields

$$[I]: \begin{array}{c} \underline{[\phi]} \\ \phi \\ \vdots \\ \underline{\gamma} \\ \surd \end{array} \qquad\qquad [E]: \frac{\phi}{p:[\phi] \quad \downarrow(p)}$$

The introduction rule $[I]$ says that in order to prove $[\phi]$ we may prove ϕ under some assumption γ, which we are free to choose[4]. Read bottom-up, $[I]$ says that if a proposition ϕ can be derived from some assumption γ then there is a proof of $[\phi]$ which no longer depends on γ. The assumption γ effectively is discharged in favour of the $[.]$ construct which only indicates its presence. Note that even though the assumption γ to be hidden by an application of $[I]$ may be inferred (in LEGO) directly from the argument derivation $f : \gamma \to \phi$, we will supply it as an explicit argument, i.e. we write $[I](\gamma, f : \gamma \to \phi)$ rather than $[I](f : \gamma \to \phi)$.

The elimination rule is stated with the help of an operator $\downarrow$ which is meant to project out the assumption hidden in a proof of $[\phi]$. It represents the first projection π_1 from above. $[E]$ proves ϕ from a proof p of $[\phi]$ if also a proof of the assumption $\downarrow(p)$ hidden in p is given. The interaction of $[I]$, $[E]$, and $\downarrow$ is governed by the equations

$$\downarrow \circ [I](\gamma\ ,\ f : \gamma \to \phi) \;=\; \gamma \qquad (12)$$

$$[E]([I](\gamma\ ,\ f : \gamma \to \phi)\ ,\ c : \gamma) \;=\; f(c) \qquad (13)$$

which state that $\downarrow$ indeed yields the assumption hidden by $[I]$ and the elimination rule $[E]$ recovers the proof hidden by $[I]$. Note, that (13) makes implicit use of (12). The reader might find it useful to compare rules $[I]$ and $[E]$ with $\supset I$ and $\supset E$ for implication.

[4]For practiacl applications it does not seem to be a restriction to confine application of $[I]$ to propositions γ of the base logic. Since it is technically advantageous our implementation imposes this restriction.

Two examples of rules which can be derived from the rules introduced are 'lifting' $[L]$ and '*constrained* $\wedge$-*introduction*' $[\wedge I]$:

$$[L] : \begin{array}{c} \dfrac{[\psi]}{\psi \quad [\phi]} \\ \vdots \\ \dfrac{\phi}{\surd} \end{array} \qquad\qquad [\wedge I] : \dfrac{[\phi \wedge \psi]}{[\phi] \quad [\psi]}$$

Rule $[L]$ lifts an unconstrained derivation $f : \phi \rightarrow \psi$ to a constrained one, namely if ϕ is provable under some (hidden) assumption then the conclusion ψ is provable under the same (hidden) assumption. Rule $[\wedge I]$ is the $\wedge$ introduction for constrained propositions melting the hidden assumptions of $[\phi]$ and $[\psi]$ into a single hidden assumption for their conjunction. This interpretation of $[L]$ and $[\wedge I]$ is captured by the equations

$$\downarrow \circ [L](f : \phi \rightarrow \psi \, , \, p : [\phi]) \;=\; \downarrow(p) \tag{14}$$

$$\downarrow \circ [\wedge I](p : [\phi] \, , \, q : [\psi]) \;=\; \downarrow(p) \wedge \downarrow(q) \tag{15}$$

That $[L]$ and $[\wedge I]$ are derived rules is shown by the derivation trees in Figure 3. The derivations translate into the following 'programs' implementing the two rules:

$$\begin{aligned} [L](f : \phi \rightarrow \psi \, , \, p : [\phi]) &\overset{df}{=} [I](\downarrow(p) \, , \, \lambda x : \downarrow(p). \supset E(\supset I(f) \, , \, [E](p, x))) \\ [\wedge I](p : [\phi] \, , \, q : [\psi]) &\overset{df}{=} [I](\downarrow(p) \wedge \downarrow(q) \, , \, \lambda x : \downarrow(p) \wedge \downarrow(q). \\ &\qquad \wedge I([E](p, \wedge E_l(x)) \, , \, [E](q, \wedge E_r(x)))) \end{aligned}$$

for which the equations (14) and (15) immediately follow from (12) above. Let us take $[L]$ as an example to explain how the program is constructed. The arguments to the $[L]$ rule is a derivation $f : \phi \rightarrow \psi$ and a proof $p : [\phi]$. The derivation tree for $[L]$ (upper tree) in Figure 3 describes which rules have to be composed in which order to transform these two arguments into a proof of $[\psi]$. The term given above for defining $[L]$ is exactly this composition. The top rule $[I]$ is applied to a pair consisting of an assumption (to be hidden in $[\psi]$), in this case $\downarrow(p)$, and a derivation of ψ from this assumption. This derivation $g : \downarrow(p) \rightarrow \psi$ is given by the subtree with 'input' $x : \downarrow(p)$ and 'output' ψ. The associated subterm for g is the λ-term $g = \lambda x : \downarrow(p). \; \supset E(\supset I(f) \, , \, [E](p, x))$. The λ-abstraction corresponds to the discharging of the assumption $x : \downarrow(p)$ by $[I]$.

We should perhaps remark at this point that in the the real proof session all of the proof terms are automatically constructed and manipulated by LEGO. The user only develops the proof tree top-down in an interactive way and does not have to be concerned with the underlying proof terms. This also applies to the equations which in LEGO are built-in automatic reductions. Thus, whenever proof terms are explicitly spelled out in the rest of the paper it is to show the mathematical mechanism of handling constraints as parts of proofs. All of this, however, is done by LEGO and need not bother the user.

Let us end this section with some remarks concerning the LEGO implementation of the logic. *In the LEGO implementation* the base logic is a higher order intuitionistic calculus encoded in the type theory of the Calculus of Constructions, or more precisely in Luo's extension $\Sigma CC_{\subset}$[22] of it. An introduction on using type theories as logical calculi can be found in [21]. The following papers present examples of encoding particular logics in

$$\dfrac{\dfrac{[\psi]}{\psi}\,[I](x)}{\begin{array}{c}\dfrac{\phi\supset\psi}{\psi}\,{\supset}I\\ \vdots\\ \dfrac{\phi}{\surd}\end{array}\quad \dfrac{\phi}{p:[\,\phi\,]\quad \dfrac{x:\downarrow(p)}{\surd}}\,[E]}\,{\supset}E$$

$$\dfrac{\dfrac{[\phi\wedge\psi]}{\phi\wedge\psi}\,[I](x)}{\dfrac{\dfrac{\phi}{p:[\phi]\quad \downarrow(p)}\,[E]}{\dfrac{x:(\downarrow(p)\wedge\downarrow(q))}{\surd}}\,{\wedge}E_l\qquad \dfrac{\dfrac{\psi}{q:[\psi]\quad \downarrow(q)}\,[E]}{(x)}\,{\wedge}E_r}\,{\wedge}I$$

Figure 3: The derived rules $[\,L\,]$ and $[\,\wedge I\,]$

the type theory of LF [24,1], CC [6], or Martin-Löf [27,28]. For implementating the $[\,.\,]$ operator it suffices to assumes a type universe *Type* which is closed under the formation of simple function types $A \to B$, dependent function types $\Pi x : A.M$, and strong sum types $\Sigma x{:}A.M$. Further, there is a type *Prop*, the type of all propositions of the base logic which is embedded in *Type* both as a type (the type of propositions) and as a subuniverse (each proposition is the type of its proofs), in symbols

$$Prop : Type \text{ and } Prop \subset Type.$$

All this is provided by the type theory $\Sigma CC_{\subset}$.

If ϕ : *Prop* and ψ : *Prop* are propositions then the proposition $\phi \supset \psi$: *Prop* is encoded as the function type $\phi \to \psi$, i.e. a proof of $\phi \supset \psi$ is a function that maps proofs of ϕ to proofs of ψ[5]. This is all what is needed for encoding Definition 1, on which the logic of constrained propositions is based, as a particular type of proofs. Definition 1 is represented by the following Σ-type:

$$[\phi] \overset{df}{=} \Sigma\gamma : Prop.\,\gamma \supset \phi.$$

A Σ-type $\Sigma a : A.B(a)$ is the type of all pairs (a, b) where a is of type A and b is of type $B(a)$. The crucial feature of a Σ-type is that the type of the second component (here $B(a)$) may depend on the value of the first (here a). This is the feature needed to express $[\,\phi\,]$ as a type: the type of pairs where the second component is a proof of a proposition $(\gamma \supset \phi)$ which

[5]Thus, in our LEGO implementation of the logic derivations $f : \phi \to \psi$ and proofs $p : \phi \supset \psi$ are actually the same

depends on the first component (the assumption γ). This is why the implementation resides on the fact that LEGO provides Σ-types[6]. The properties of constrained propositions then follow from the properties of Σ types within the LEGO type system.

Now, having available a logic for dealing with constrained propositions, we turn back to the example and demonstrate its use.

4 Verification of the Example

To keep explanations reasonably short we will not mention all details that have to be presented in the actual (complete) LEGO implementation. Among those are for instance all definitions to do with basic data types such as integers and booleans. We simply assume a LEGO context in which the usual mathematical properties regarding these data types are available in the base logic. We may then focus on those parts of the verification that are done using the constrained logic introduced in the previous section. The example presented has been completely formalised and verified in LEGO.

The task, set up in Section 2.2, is to find a derivation for

$$\exists z : \mathit{signal}.\ \mathit{cnt\_abs}\,(x,y,z)\ \vdash\ \mathit{cnt\_appr}\,(x,y) \tag{16}$$

where

$$\begin{aligned}
\mathit{xor\_abs}\,(x,y,z) &= (\mathit{stable}\,x\,\delta_{xor}\ \wedge\ \mathit{stable}\,y\,\delta_{xor}) \\
&\qquad \supset\ \forall t : \mathit{int}.\ \mathit{tick}\ t\ \supset\ zt = xt + yt \\
\mathit{latch\_abs}\,(d,q) &= (\mathit{one\_shot}\ \wedge\ \mathit{min\_sep}\,\delta_{latch})\ \supset\ \forall t_1, t_2 : \mathit{int}. \\
&\qquad\qquad (\mathit{tick}\ t_1\ \wedge\ \mathit{tick}\ t_2)\ \supset\ (\mathit{next\_abs}(t_1\,,\,t_2)\ \supset\ q(t_2) = d(t_1)) \\
\mathit{cnt\_abs}\,(x,y,z) &= \mathit{xor\_abs}\,(x,z,y)\ \wedge\ \mathit{latch\_abs}\,(y,z) \\
\mathit{cnt\_appr}\,(x,y) &= (C_0\ \wedge\ C_1(x,y))\ \supset\ (\forall t_1, t_2 : \mathit{int}.\ C_2(t_1,t_2) \\
&\qquad \supset\ (\mathit{next\_abs}\,(t_1,t_2)\ \supset\ y(t_2) = x(t_2) + y(t_1)))
\end{aligned}$$

which is split into a main proof, free of constraints, and a successive constraint analysis to establish constraints C_0, C_1, C_2 for the composite device. The first goal is achieved by reformulating *xor_abs*, *latch_abs*, *cnt_abs*, and *cnt_appr* as constrained propositions:

$$\begin{aligned}
\mathit{xor\_abs'}\,(x,y,z) &\stackrel{df}{=} [\forall t : \mathit{int}.[\,zt = xt + yt\,]] \\
\mathit{latch\_abs'}\,(d,q) &\stackrel{df}{=} [\forall t_1, t_2 : \mathit{int}.[\,\mathit{next\_abs}\,(t_1\,,\,t_2)\ \supset\ q(t_2) = q(t_1)\,]] \\
\mathit{cnt\_abs'}\,(x,y,z) &\stackrel{df}{=} \mathit{xor\_abs'}(x,z,y)\ \wedge\ \mathit{latch\_abs'}(y,z) \\
\mathit{cnt\_appr'}\,(x,y) &\stackrel{df}{=} [\forall t_1, t_2 : \mathit{int}.[\,\mathit{next\_abs}\,(t_1\,,\,t_2)\ \supset\ y(t_2) = x(t_2) + y(t_1)\,]]
\end{aligned}$$

Syntactically speaking, all constraints are now removed from the formulae and replaced by the [ ] construct. Semantically speaking, and this is the crucial idea, a constraint now is no longer part of the proposition but of the proof. For instance,

$$[\forall t : \mathit{int}.[\,zt = xt + yt\,]] \tag{17}$$

[6]The current implementation also makes use of LEGO's type universes and the built-in synthesis of universe levels, also called "typical ambiguity".

does not give any more information regarding constraints than indicating that there *may be* hidden assumptions, namely one for each instance of the [ ]-operator. It is the proof of (17) that actually determines these constraints. In fact, the constraints depend on from which low-level axioms about the exclusive-or gate the abstracted proposition (17) is derived, and by which abstraction process. Here *xor* is used but one might take a more detailed description of the gate, e.g. with variable delays, and then of course some other constraints would result. Also, there may me more than one way to verify an abstract behaviour of a composite device from properties about its components and each may result in a different constraint.

4.1 Abstract Verification of Modulo-2 Counter

As the 'constraint-free' version of (16) we now demonstrate a derivation of

$$\exists z : \mathit{signal}.\ \mathit{cnt\_abs}'(x,y,z) \ \vdash\ \mathit{cnt\_appr}'(x,y) \tag{18}$$

It differs from the ideal derivation (4) only in the presence of the [.]-construct.

First note that we can dispense with the existential quantifier since (18) is logically equivalent to a derivation

$$\mathit{cnt\_abs}'(x,y,z) \ \vdash\ \mathit{cnt\_appr}'(x,y) \tag{19}$$

i.e. every proof of (18) gives rise to a proof of (19) and vice versa. Thus it does not really matter whether we hide internal signals in the implementation (left hand side of the turnstile) in the first place or not; they are always eventually 'opened' when verified against a specification (right hand side). Consequently, we decide to take (19) as the verification goal right away.

Now let us introduce the following syntactic abbreviations:

$$\begin{array}{rcl} \theta & \overset{df}{\equiv} & \mathit{next\_abs}\,(t_1\,,\,t_2) \\ \epsilon & \overset{df}{\equiv} & y(t_2) = x(t_2) + y(t_1) \\ \phi & \overset{df}{\equiv} & yt = xt + zt \\ \psi & \overset{df}{\equiv} & z(t_2) = y(t_1) \end{array}$$

where x, y, z are, from now on, fixed variables of type *signal* and t_1, t_2 fixed variables of type *int*. With these abbreviations (19) essentially amounts to finding a derivation

$$\frac{[\forall t_1, t_2 : \mathit{int}.[\theta \supset \epsilon]]}{[\forall t : \mathit{int}.[\phi]] \ \wedge\ [\forall t_1, t_2 : \mathit{int}.[\theta \supset \psi]]}$$

This will depend on a derivation

$$\mathbf{Q}\ :\ (\phi\{t_2/t\}\ \wedge\ \psi) \to \epsilon$$

which shall be assumed given.

Figure 4 shows the complete natural deduction tree for (19), using $\mathbf{Q}$ and applying the rules of Section 3, where all typing information is removed to improve legibility. The refinement mechanism in LEGO allows constructing the derivation and the corresponding

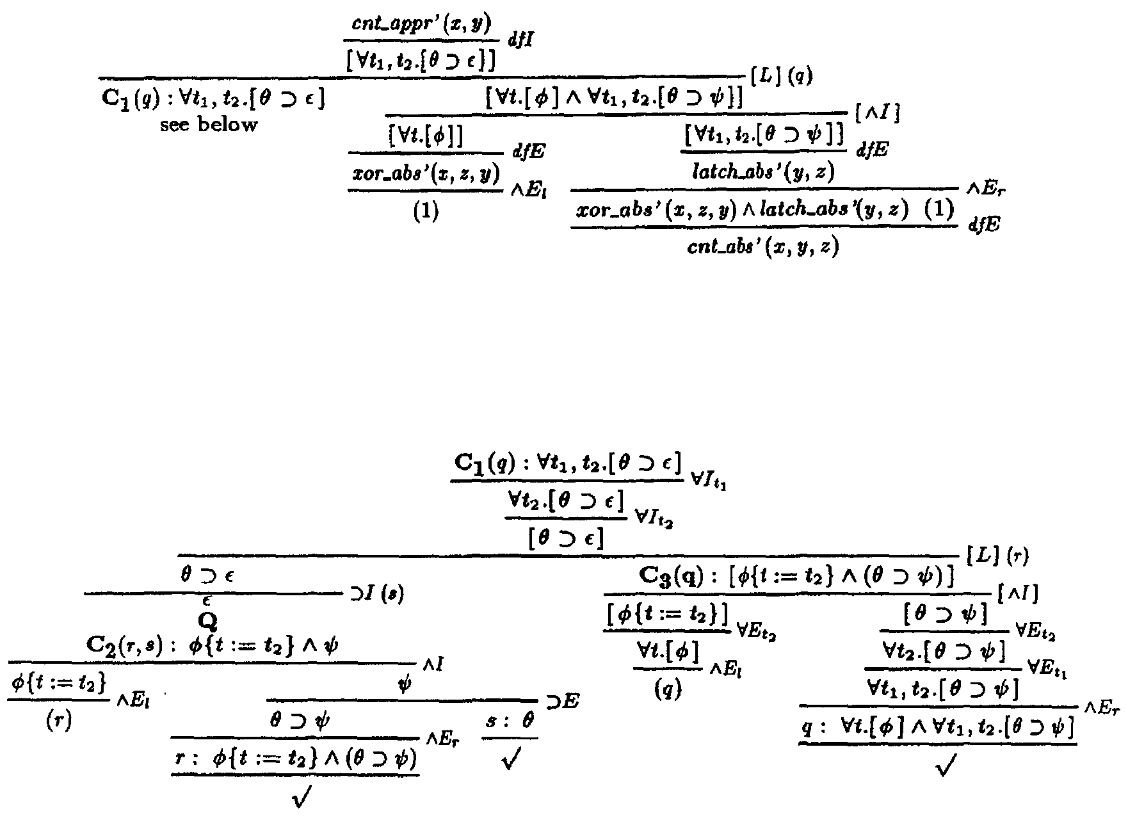

Figure 4: Verification of correctness for *modulo*-2 counter

proof term interactively in a top-down fashion. The proof term collects together the rules in the order in which they are applied. In this case the proof tree of Figure 4 defines the proof function

$$\mathbf{C} : \textit{cnt\_abs'}(x,y,z) \rightarrow \textit{cnt\_appr'}(x,y)$$

which (if typing information is again supressed) reads:

$$\begin{array}{rcl} \mathbf{C}(p_C) & = & dfI \circ [L](\lambda q.\mathbf{C}_1(q)\,,\,[\wedge I](dfE \circ \wedge E_l \circ dfE(p_C)\,,\,dfE \circ \wedge E_r \circ dfE(p_C))) \\ \mathbf{C}_1(q) & = & \forall I_{t_1} \circ \forall I_{t_2} \circ [L](\lambda r.\supset I(\lambda s.\mathbf{Q} \circ \mathbf{C}_2(r,s))\,,\,\mathbf{C}_3(q)) \\ \mathbf{C}_2(r,s) & = & \wedge I(\wedge E_l(r)\,,\,\supset E(\wedge E_r(r)\,,\,s)) \\ \mathbf{C}_3(q) & = & [\wedge I](\forall E_{t_2} \circ \wedge E_l(q)\,,\,\forall E_{t_2} \circ \forall E_{t_1} \circ \wedge E_r(q)) \end{array}$$

This derivation provides a solution to the first part of the task: It corresponds to the main proof that composing a delay-free *modulo*-2 sum and an one-unit delay as in Figure 2 yields a stoppable *modulo*-2 counter. The only steps in the above derivation that would not arise in an ideal proof, i.e. one which does not consider constraints at all, are the two occurrences of the $[L]$ rule. This rule is necessary to lift a proof from the base logic, for instance $\mathbf{Q}$, to a constrained proof.

The derivation can be called abstract since it is performed without looking at or even manipulating explicit constraints. Remember that *cnt_abs'* and the other formulae do not actually carry constraints. The square brackets only indicate where constraints are to be expected and so intuitively serve as a ***place-holder*** for constraints. In manipulating the place-holder instead of real constraints the (abstract) derivation $\mathbf{C}$ is independent of constraints and yet retains enough information for extracting the constraints inside $\mathbf{C}(p_C)$: *cnt_appr'*(x,y) out of those in p_C : *cnt_abs'*(x,y,z), which can now be done in a completely separate phase: in the *constraint analysis.*

Before we can demonstrate this we need to give proofs

$$\begin{array}{rcl} \mathbf{X} & : & \textit{xor}\,(x,z,y) \rightarrow \textit{xor\_abs'}(x,z,y) \\ \mathbf{L} & : & \textit{latch}\,(y,clk,z) \rightarrow \textit{latch\_abs'}(y,z) \end{array}$$

and thus establish *that* and *how* the exclusive-or and latch are implementations of the abstract components. In these proofs the actual constraints for the place-holders inside *xor_abs'* and *latch_abs'* and consequently inside

$$\textit{cnt\_abs'}(x,y,z) = \textit{xor\_abs'}(x,z,y) \;\wedge\; \textit{latch\_abs'}(y,z)$$

will be determined.

4.2 Synchronous Abstraction of Exclusive-Or and Latch

We begin with the derivation $\mathbf{X}$. The main means for introducing constraints of course is the $[I]$ rule which will be used twice, namely for hiding the constraints *stable* $x\,\delta_{xor}$ $\wedge$ *stable* $z\,\delta_{xor}$ and ***tick*** t. The derivation will also have to assume $\delta_{xor} \geq 0$, use substitution rules -all abbreviated as '*subst*'-, and the following facts about *int*:

$$Inv : \frac{\forall t,u : int.\ t - u + u = t}{\top} \qquad Refl : \frac{\forall t : int.\ t \leq t}{\top} \qquad Sub : \frac{\forall t,u : int.\ u \geq 0 \supset t - u \leq t}{\top}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{xor\_abs'(x,z,y)}{[\forall t.[yt = xt + zt]]}\,dfI}{\forall t.[yt = xt + zt]}\,[I]\,(p)}{[yt = xt + zt]}\,\forall I_t}{yt = xt + zt}\,[I]\,(q)}{\dfrac{yt = x(t-\delta_{xor}) + zt}{\mathbf{X_1}(p,q) : yt = x(t-\delta_{xor}) + z(t-\delta_{xor}) \quad \mathbf{X_4}(p,q) : zt = z(t-\delta_{xor})}\,\text{subst} \qquad \mathbf{X_2}(p,q) : xt = x(t-\delta_{xor})}\,\text{subst}$$

$$\dfrac{\mathbf{X_1}(p,q) : yt = x(t-\delta_{xor}) + z(t-\delta_{xor})}{\dfrac{y(t-\delta_{xor}+\delta_{xor}) = yt}{\dfrac{t-\delta_{xor}+\delta_{xor} = t}{\dfrac{\forall u.\ t-u+u = t}{\dfrac{\forall t,u.\ t-u+u = t}{\top}\,\text{Inv}}\,\forall E_t}\,\forall E_{\delta_{xor}}}\,\text{subst} \qquad \dfrac{y(t-\delta_{xor}+\delta_{xor}) = x(t-\delta_{xor}) + z(t-\delta_{xor})}{\dfrac{\forall t.\ y(t+\delta_{xor}) = xt + zt}{xor\,(x,z,y)}\,dfE}\,\forall E_{t-\delta_{xor}}}\,\text{subst}$$

$$\dfrac{\mathbf{X_2}(p,q) :\ xt = x(t-\delta_{xor})}{\dfrac{(tick\,t \wedge t-\delta_{xor} \leq t-\delta_{xor} \leq t) \supset xt = x(t-\delta_{xor})}{\dfrac{\forall t_2.(tick\,t \wedge t-\delta_{xor} \leq t_2 \leq t) \supset xt = x(t_2)}{\dfrac{\forall t_1,t_2.(tick\,t_1 \wedge t_1-\delta_{xor} \leq t_2 \leq t_1) \supset x(t_1) = x(t_2)}{\dfrac{stable\ x\ \delta_{xor}}{\dfrac{p :\ stable\ x\ \delta_{xor} \wedge stable\ z\ \delta_{xor}}{\surd}}\,\wedge E_r}\,dfE}\,\forall E_t}\,\forall E_{t-\delta_{xor}} \qquad \mathbf{X_3}(q) : tick\,t \wedge t-\delta_{xor} \leq t-\delta_{xor} \leq t}\,\supset E$$

$$\dfrac{\mathbf{X_3}(q) :\ tick\,t \wedge t-\delta_{xor} \leq t-\delta_{xor} \leq t}{\dfrac{q :\ tick\,t}{\surd} \qquad \dfrac{t-\delta_{xor} \leq t-\delta_{xor} \leq t}{\dfrac{t-\delta_{xor} \leq t-\delta_{xor} \wedge t-\delta_{xor} \leq t}{\dfrac{t-\delta_{xor} \leq t-\delta_{xor}}{\dfrac{\forall t.\,t \leq t}{\top}\,\text{Refl}}\,\forall E_{t-\delta_{xor}} \qquad \dfrac{t-\delta_{xor} \leq t}{\dfrac{\delta_{xor} \geq 0 \supset t-\delta_{xor} \leq t}{\dfrac{\forall u.\,u \geq 0 \supset t-u \leq t}{\dfrac{\forall t,u.\,u \geq 0 \supset t-u \leq t}{\top}\,\text{Sub}}\,\forall E_t}\,\forall E_{\delta_{xor}} \qquad \delta_{xor} \geq 0}\,\supset E}\,\wedge I}\,dfI}\,\wedge I$$

Figure 5: Derivation for abstracting *xor* as a synchronous device

Figure 5 presents the proof tree, again split into several parts. The subtree named $\mathbf{X_4}(p,q)$ is not given as it is identical to the subtree $\mathbf{X_2}(p,q)$ with variable x is replaced by z and the $\wedge E_l$ rule instead of $\wedge E_r$.

The two constraints built into the proof term by the introduction rule $[I]$ can be extracted by applying the $\downarrow$ function to the proof term $\mathbf{X}$ in the appropriate places. To this end let p_X be a proof of *xor* (x,z,y) and consider the proof term $\mathbf{X}(p_X)$ of *xor_abs'* (x,z,y):

$$\begin{aligned}
\mathbf{X}(p_X) &= dfI \circ [I](stable\, x\, \delta_{xor} \;\wedge\; stable\, z\, \delta_{xor} \;, \\
&\qquad \lambda p.\forall I_t \circ [I](tick\, t \;,\; \lambda q.subst(subst(\mathbf{X_1}(p,q)(p_X) \;,\; \mathbf{X_4}(p,q)) \;,\; \mathbf{X_2}(p,q)))) \\
\mathbf{X_1}(p,q)(p_X) &= subst(subst \circ \forall E_{\delta_{xor}} \circ \forall E_t \circ Inv \;,\; \forall E_{t-\delta_{xor}} \circ dfE(p_X)) \\
\mathbf{X_2}(p,q) &= \supset E(\forall E_{t-\delta_{xor}} \circ \forall E_t \circ dfE \circ \wedge E_r(p) \;,\; \mathbf{X_3}(q)) \\
\mathbf{X_4}(p,q) &= \supset E(\forall E_{t-\delta_{xor}} \circ \forall E_t \circ dfE \circ \wedge E_l(p) \;,\; \mathbf{X_3}(q)) \\
\mathbf{X_3}(q) &= \wedge I(q \;,\; dfI \circ \wedge I(\forall E_{t-\delta_{xor}} \circ Refl \;,\; \forall E_{\delta_{xor}} \circ \forall E_t \circ Sub))
\end{aligned}$$

Now from

$$dfE \circ \mathbf{X}(p_X) : [\forall t.[\, yt = xt + zt\,]]$$

we can regain the constraint hidden by the outer ocurrence of the square brackets by computing

$$\begin{aligned}
CX_{outer}(x,z,y) &\overset{df}{=} \downarrow \circ\, dfE \circ \mathbf{X}(p_X) \\
&= \downarrow \circ\, dfE \circ dfI \circ [I](stable\, x\, \delta_{xor} \wedge stable\, z\, \delta_{xor} \;,\; \lambda p.\cdots) \\
\text{by (11)} \quad &= \downarrow \circ [I](stable\, x\, \delta_{xor} \;\wedge\; stable\, z\, \delta_{xor} \;,\; \lambda p.\cdots) \\
\text{by (12)} \quad &= stable\, x\, \delta_{xor} \;\wedge\; stable\, z\, \delta_{xor}
\end{aligned}$$

In order to get to the constraint hidden by the inner occurrence of $[\,.\,]$ we have to assume a proof $p : stable\, x\, \delta_{xor} \;\wedge\; stable\, z\, \delta_{xor}$ of the outer constraint and a variable t to instantiate the universal quantifier with. Then,

$$\forall E_t \circ [E](dfE \circ \mathbf{X}(p_X) \;,\; p) : [\, yt = xt + zt\,]$$

and

$$\begin{aligned}
CX_{inner}(x,z,y;t) &\overset{df}{=} \downarrow \circ \forall E_t \circ [E](dfE \circ \mathbf{X}(p_X) \;,\; p) \\
&= \downarrow \circ \forall E_t \circ [E](dfE \circ dfI \circ [I](stable\, x\, \delta_{xor} \;\wedge\; stable\, z\, \delta_{xor} \;,\; \lambda p.\cdots) \;,\; p) \\
\text{by (11)} \quad &= \downarrow \circ \forall E_t \circ [E]([I](stable\, x\, \delta_{xor} \;\wedge\; stable\, z\, \delta_{xor} \;,\; \lambda p.\cdots) \;,\; p) \\
\text{by (13)} \quad &= \downarrow \circ \forall E_t \circ \lambda p.\cdots (p) \\
&= \downarrow \circ \forall E_t(\cdots) \\
&= \downarrow \circ \forall E_t(\forall I_t([I](tick\; t \;,\; \lambda q.\cdots))) \\
&= \downarrow \circ \forall E_t \circ \forall I_t([I](tick\, t \;,\; \lambda q.\cdots)) \\
\text{by (10)} \quad &= \downarrow \circ [I](tick\, t \,,\, \lambda q.\cdots)) \\
\text{by (12)} \quad &= tick\, t
\end{aligned}$$

These computations show that, modulo renaming of variables,

$$\mathbf{X} : xor\,(x,z,y) \longrightarrow xor\_abs'\,(x,z,y)$$

indeed captures the derivation

$$xor\,(x,y,z) \;\vdash\; xor\_abs\,(x,y,z)$$

with the difference that the constaints do not show up in the proposition *xor_abs'* of abstract behaviour but in the proof **X**. Similarly, there is a derivation

$$\mathbf{L} : latch\,(y, clk, z) \longrightarrow latch\_abs'(y,z)$$

for the latch swallowing the constraints in *latch_abs*, which can also be constructed interactively in LEGO. It is slightly more involved than **X** for it has to utilize bounded induction on *int*. We do not present the derivation here and content ourselves with stating that it contains the expected constraints. More precisely, if p_L is a proof of *latch* (y, clk, z) then

$$dfE \circ \mathbf{L}(p_L) \;:\; [\forall t_1, t_2 : int.[\,next\_abs\,(t_1\,,\,t_2) \;\supset\; q(t_2) = q(t_1)\,]\,]$$

and we have

$$\begin{array}{rcl} CL_{outer}(y,z) & \overset{df}{=} & \downarrow \circ\, dfE \circ \mathbf{L}(p_L) \\ & = & one\_shot \;\wedge\; min\_sep\, \delta_{latch} \end{array}$$

and for q a proof of $one\_shot \wedge min\_sep\, \delta_{latch}$, and t_1, t_2 variables of type *int*

$$\begin{array}{rcl} CL_{inner}(y,z;t_1,t_2) & \overset{df}{=} & \downarrow \circ \forall E_{t_2} \circ \forall E_{t_1} \circ [\,E\,](dfE \circ \mathbf{L}(p_L)\;,\;q) \\ & = & tick\,t_1 \;\wedge\; tick\,t_2 \end{array}$$

4.3 Constraint Analysis

Section 4.1 has shown that composition of delay-free *modulo*-2 sum and one-unit delay satisfies the specification of a *modulo*-2 counter. The verification is summed up in the derivation

$$\mathbf{C} : cnt\_abs'(x,y,z) \longrightarrow cnt\_appr'(x,y) \tag{20}$$

In the previous section we gave proofs

$$\begin{array}{rcll} \mathbf{X} & : & xor\,(x,z,y) \longrightarrow xor\_abs'(x,z,y) & (21) \\ \mathbf{L} & : & latch\,(y, clk, z) \longrightarrow latch\_abs'(y,z) & (22) \end{array}$$

witnessing that in a synchronous environment *xor* and *latch* may be regarded as implementations of the corresponding abstract components delay-free *modulo*-2 sum and one-unit delay. Residing inside **X** and **L** are certain behavioural constraints which record assumptions about the environment of the components under which this abstraction is possible. Now we may put pieces together and prove that in a synchronous environment the composition of *xor* and *latch* can be regarded as an implementation of the abstract *modulo*-2 counter. This comes down to a derivation

$$\mathbf{I} : xor\,(x,z,y) \;\wedge\; latch\,(y, clk, z) \;\longrightarrow\; cnt\_appr'(x,y)$$

from the data (20), (21), and (22).

$$\begin{array}{c}
\mathit{cnt\_appr}'(x,y)\\
\mathbf{C}\\
\dfrac{\mathit{cnt\_abs}'(x,y,z)}{\dfrac{\mathit{xor\_abs}'(x,z,y)\wedge \mathit{latch\_abs}'(y,z)}{\begin{array}{cc} \mathit{xor\_abs}'(x,z,y) & \mathit{latch\_abs}'(y,z)\\ \mathbf{X} & \mathbf{L}\\ \dfrac{\mathit{xor}(x,z,y)}{\mathit{xor}(x,z,y)\wedge \mathit{latch}(y,\mathit{clk},z)\ (1)}\wedge E_l & \dfrac{\mathit{latch}(y,\mathit{clk},z)}{(1)}\wedge E_r\end{array}}\wedge I}\ \mathit{dfI}
\end{array}$$

Figure 6: Low-level implementation of *modulo*-2 counter

As an aside note that again all applications of existential quantification for hiding internal signals are dropped. This is done without loss of generality since it can be shown that taking explicit account of hiding would reduce to the case considered here anyway. A more detailed discussion of hiding in the context of the proposed logic is outside the scope of this paper but we believe that hiding of signals via existential quantification does not introduce any intrinsic complications. After all, hiding seems to be eliminable for all practical examples.

Constraint analysis consists in examining the constraints residing in **I** which, as they are synthesised from the constraints in **X** and **L**, will embody the weakest constraint for the composite circuit which guarantees that the constraints of its components are met. Figure 6 shows the derivation tree for **I**. The corresponding proof term reads as follows:

$$\begin{aligned}
\mathbf{I}(p_I) &= \mathbf{C}\circ \mathit{dfI}\circ \mathbf{I}_1(p_I)\\
\mathbf{I}_1(p_I) &= \wedge I(\mathbf{X}\circ \wedge E_l(p_I)\ ,\ \mathbf{L}\circ\wedge E_r(p_I))
\end{aligned}$$

where p_I is a proof of $\mathit{xor}(x,z,y)\wedge \mathit{latch}(y,\mathit{clk},z)$. From

$$\mathit{dfE}\circ \mathbf{I}(p_I) : [\forall t_1,t_2.[\mathit{next\_abs}(t_1\ ,\ t_2)\ \supset\ y(t_2)=x(t_2)+y(t_1)\,]\,]$$

we may then extract constraints for both occurrences of [ ]. The most important of these is the outer one:

$$\begin{array}{rcl}
CI_{outer}(x,y,z) & \overset{df}{=} & \downarrow\circ \mathit{dfE}\circ \mathbf{I}(p_I)\\
 & = & \downarrow\circ \mathit{dfE}\circ \mathbf{C}\circ \mathit{dfI}\circ \mathbf{I}_1(p_I)\\
 & = & \downarrow\circ \mathit{dfE}\circ \mathit{dfI}\circ [\,L\,](\lambda q.\ \mathbf{C}_1(q)\ ,\\
 & & [\wedge I\,](\mathit{dfE}\circ\wedge E_l\circ \mathit{dfE}\circ \mathit{dfI}\circ \mathbf{I}_1(p_I)\ ,\ \mathit{dfE}\circ\wedge E_r\circ \mathit{dfE}\circ \mathit{dfI}\circ \mathbf{I}_1(p_I)))\\
\text{by (11)} & = & \downarrow\circ[\,L\,](\lambda q.\ \mathbf{C}_1(q)\ ,\ [\wedge I\,](\mathit{dfE}\circ\wedge E_l\circ \mathbf{I}_1(p_I)\ ,\ \mathit{dfE}\circ\wedge E_r\circ \mathbf{I}_1(p_I)))\\
\text{by (14)} & = & \downarrow\circ[\wedge I](\mathit{dfE}\circ\wedge E_l\circ \mathbf{I}_1(p_I)\ ,\ \mathit{dfE}\circ\wedge E_r\circ \mathbf{I}_1(p_I)))\\
\text{by (15)} & = & \downarrow\circ \mathit{dfE}\circ\wedge E_l\circ \mathbf{I}_1(p_I)\ \wedge\ \downarrow\circ \mathit{dfE}\circ\wedge E_r\circ \mathbf{I}_1(p_I)\\
 & = & \downarrow\circ \mathit{dfE}\circ\wedge E_l\circ\wedge I(\mathbf{X}\circ\wedge E_l(p_I)\ ,\ \mathbf{L}\circ\wedge E_r(p_I))\\
 & & \wedge\ \downarrow\circ \mathit{dfE}\circ\wedge E_r\circ\wedge I(\mathbf{X}\circ\wedge E_l(p_I)\ ,\ \mathbf{L}\circ\wedge E_r(p_I))\\
 & = & \downarrow\circ \mathit{dfE}\circ \mathbf{X}\circ\wedge E_l(p_I)\ \wedge\ \downarrow\circ \mathit{dfE}\circ \mathbf{L}\circ\wedge E_r(p_I)\\
 & = & \mathit{stable}\ x\ \delta_{xor}\ \wedge\ \mathit{stable}\ z\ \delta_{xor}\ \wedge\ \mathit{one\_shot}\ \wedge\ \mathit{min\_sep}\ \delta_{latch}
\end{array}$$

The inner constraint on time points recorded within $\mathit{dfE}\circ \mathbf{I}(p_I)$ can be computed as

$$CI_{inner}(x,y,z;t_1,t_2)\ =\ \mathit{tick}\ t_2\ \wedge\ \mathit{tick}\ t_1\ \wedge\ \mathit{tick}\ t_2$$

This shows that the derivation I has in fact collected together the constraints for *xor* and *latch*.

Constraint analysis in this framework is ***proof analysis***: 'analyse the constraint hidden in a constrained proof and replace it by a simpler proposition'. Consider the general situation of a derivation $f : \phi \rightarrow [\psi]$, of which I is a special instance. f constructs for each proof p of ϕ essentially a pair consisting of a hidden assumption γ and a proof of $\gamma \supset \psi$.[7] Performing a constraint analysis on f means replacing γ by a *simpler* or *weaker* assumption γ'. The condition under which this is possible is that

$$\phi \wedge \gamma' \vdash \gamma$$

holds, i.e. γ' together with ϕ is stronger than γ. (Note, this means γ can always be replaced by a *stronger* assumption $\gamma' \vdash \gamma$). In the extreme case where γ' is to be the weakest possible assumption, namely $\gamma' = \top$, this amounts to proving γ from ϕ. Given that the hidden assumption γ is an input constraint of a hardware device this will only be possible if ϕ contains complete information about the environment of the device and then amount to proving that the environment satisfies the input constraints. The typical case, however, is that ϕ (as in I) merely describes parts of a complete circuit in which case only 'parts' of γ will follow from ϕ while other 'parts' have to be retained in γ'. Formally, constraint analysis may be summarised by the following

Proposition 1 *Let $f : \phi \rightarrow [\psi]$ be a derivation in the logic of constrained proofs and γ the constraint constructed by f, i.e. γ is a proposition of the base logic with $\downarrow \circ f(p) = \gamma$ for all $p : \phi$. Then for all propositions γ' of the base logic such that $\phi \wedge \gamma' \vdash \gamma$ there is a derivation $f' : \phi \rightarrow [\psi]$ with $\downarrow \circ f'(p) = \gamma'$ for all $p : \phi$.*

Back to the example: Since we regard signal z as internal to the composite circuit we want to replace $CI_{outer}(x, y, z)$ by a constraint of the form $C_0 \wedge C_1(x, y)$ which does not have z as a free variable and thus embodies a constraint on the environment of the *modulo*-2 counter. In view of the above discussion on constraint analysis we are lead to search for a proof of

$$xor\,(x, z, y) \wedge latch\,(y, clk, z) \wedge C_0 \wedge C_1(x, y) \vdash CI_{outer}(x, y, z)$$

It is not difficult to see that this is equivalent to a derivation

$$C_0 \wedge C_1(x, y) \vdash \forall z.\,(xor\,(x, z, y) \wedge latch\,(y, clk, z)) \supset CI_{outer}(x, y, z)$$

or equivalently

$$\begin{aligned} C_0 \wedge C_1(x, y) \;\vdash\; & one\_shot \wedge min\_sep\; \delta_{latch} \\ & \wedge \forall z.\,(xor\,(x, z, y) \wedge latch\,(y, clk, z)) \supset (stable\; x\; \delta_{xor} \wedge stable\; z\; \delta_{xor}) \end{aligned} \tag{23}$$

A simple way to arrive at constraints C_0 and C_1 that satisfy this property is simply to use the right hand side of the derivation as their definition, i.e.

$$\begin{aligned} C_0 &\overset{df}{=} one\_shot \wedge min\_sep\; \delta_{latch} \\ C_1(x, y) &\overset{df}{=} \forall z.\,(xor\,(x, z, y) \wedge latch\,(y, clk, z)) \supset (stable\; x\; \delta_{xor} \wedge stable\; z\; \delta_{xor}) \end{aligned}$$

[7] γ may in general depend on the proof p. However, it is a property of the logic that if ϕ is a proposition of the base logic then $\downarrow \circ f(p)$ must be independent of p, i.e. $\downarrow \circ f(p) = \gamma$ for some fixed proposition γ of the base logic. In particular this is the case for our example I as the computation above confirmed.

$C_1(x,y)$ says that no matter what internal signal z is produced by the circuit for given observable signals x, y the stability constraints for *xor* and *latch* are satisfied. C_0 is the restriction on the clock originating from *latch*. This choice of C_0 and C_1 corresponds to the weakest restriction on the *environment* of the composite circuit for which the constraints of all components are met.

Alternatively, a more intelligent constraint analysis could take advantage of knowledge about the type of constraints and the behaviour of the components involved and try to simplify C_0 and C_1. For instance, the stability constraint $\mathit{stable}\, z\, \delta_{xor}$ on signal z can be traded against the clock period since z is the output of the latch and the characteristic feature of the latch is to keep the output stable as long as it is not triggered by clock ticks. More precisely, we have

$$\mathit{min\_sep}\,(\delta_{latch} + \delta_{xor}) \wedge \mathit{latch}\,(y, \mathit{clk}, z) \vdash \mathit{stable}\ z\ \delta_{xor} \tag{24}$$

which suggests to define the constraints as follows

$$\begin{aligned} C_0 &\overset{df}{=} \mathit{one\_shot} \wedge \mathit{min\_sep}(\delta_{latch} + \delta_{xor}) \\ C_1(x,y) &\overset{df}{=} \mathit{stable}\ x\ \delta_{xor} \end{aligned}$$

The constraint analysis now consist in formally verifying (23). This simple proof, which we do not give here, makes reference to fact (24) and may, according to Proposition 1, be used for building a new derivation

$$\mathbf{I}' : \mathit{xor}\,(x,z,y) \wedge \mathit{latch}\,(y, \mathit{clk}, z) \rightarrow \mathit{cnt\_appr'}\,(x,y)$$

such that $\downarrow \circ\, \mathit{dfE} \circ \mathbf{I}'(p_I) = C_0 \wedge C_1(x,y)$.

5 Conclusion and Future Work

As has been hinted at in the beginning, the global aim of our work is investigating the application of abstraction mechanisms in hardware verification. This paper focused on a particular problem arising from such an undertaking, namely the problem of constraint handling. Recognising the special rôle of constraints in the design process we are proposing to distinguish at the level of the logical inference system between propositions pertaining to specifications and those pertaining to constraints. To this end a logic of constrained proofs has been introduced that, roughly speaking, provides a mechanism to move parts of a proposition into the proof. We have demonstrated by means of a simple example from synchronous circuit design that this mechanism, when used to take constraints out of specifications of abstracted behaviour, acommodates for uncoupling the verification of abstracted behaviour from the analysis of constraints without giving up the rigour of a complete formal verification. Moreover, the logic does not prejudice the type of constraints, so that its application is not limited to the usual input constraints of hardware components but also encompasses constraints on sampling times and any other type which need not be related to input signals. Also, it does not prejudice the way components are modelled, i.e. it is applicable for both the "components as functions" and the "components as predicates" paradigm.

The logic has been implemented on the interactive proof editor LEGO. First verification examples, one of which was described in the paper indicate that at least for the special case of *timing abstraction* in synchronous hardware design the implemented scheme of a constrained logic works up to the expectations. The examples are however still too simple to judge practical utility for 'real' verification problems. Also we did not as yet systematically explore the analysis of constraints within the logic and the possibility for automating parts of it in special cases like synchronous abstraction. A lot of work is left to be done here.

LEGO has proven to be a convenient and flexible environment for experimenting with a prototype logic. The scheme is implemented using $\Sigma CC_{\subset}$ and type universes, a rather powerful type theory supported by LEGO. Since the goal is to arrive at a logic that knows to differenciate between constraints and abstract properties it was important for the first experiments that LEGO does not enforce the use of any particular logic. It is basically a tool for implementing mathematics and therefore allows to experiment with various logics and to translate very directly mathematical definitions into an executable formal system. An implementation on other verification systems which are tailored to the needs of hardware design and provide the necessary infrastructure to run larger examples is planned. In particular LAMBDA [9] and VERITAS [15] seem promising candidates. LAMBDA, for instance, because its logic kernel is a higher order proof system, i.e. it manipulates rules rather than propositions, has already built-in the flexibility to introduce constraints at any time in the design process and arbitrarily to defer their analysis. The possibility of programming complex refinement tactics will allow automating large portions of constraint analysis and verification for specific circuit design styles like synchronous or speed independent circuits.

The main characteristic of the proposed logic for handling constraints is that it considers constraints as part of the proof and consequently constraint analysis as proof analysis. Although this is intuitively appealing and seems to encode the idea of decoupling abstract reasoning and constraint analysis quite well from a pragmatic point of wiew, it has to be further justified through both practical examples and mathematical analysis. The implementation in its current form is mathematically not yet completely satisfactory. For example, among all constrained proofs of a proposition $[\phi]$ there are also worthless ones, like

$$(\mathit{false}\,,\ \mathit{ex\text{-}falso\text{-}quodlibet}(\phi))$$

or

$$(\phi\,,\ \supset\! E(\lambda a : \phi.\ a)).$$

and no built-in measures have been taken to prevent these from being introduced in a proof. So far the only way to avoid this loss of information, is to restrict proofs to chose from only a well-defined set of elementary proof rules, which are known to be nicely behaved in this respect.

We consider the logic of constrained proofs an intermediate stage towards abstraction which means reasoning on (two ore more) different levels: on a concrete and an abstract level, connected via an abstraction function or relation. The example used in this paper lives at the concrete level only. For example, there is only a single notion of time, i.e. no distinction between abstract synchronous time and concrete level time. Eventually we want to combine constrained logic with abstraction mappings and reason about hardware at several levels of abstraction within one verification framework. It could be investigated in LEGO using Σ-type which provide a mechanism for theory abstraction [33] allowing to identify and keep apart the corresponding levels of theories. This was yet another reason for chosing LEGO.

6 Acknowledgements

The development of our work has been strongly influenced by discussions with Rod Burstall. We would also like to thank Mike Fourman for his encouraging support, Terry Stroup and the anonymous referees for valuable comments on a draft of this paper, and Julian Bradfield for his help with type-setting and for supplying TEX-macros for proof trees.

References

[1] A. Avron, F. Honsell, and I. Mason. Using typed lambda calculus to implement formal systems on a machine. Technical Report ECS-LFCS-87-31, Edinburgh Univ., Dept. of Comp. Sci., June 1987.

[2] J. C. Barros and B. W. Johnson. Equivalence of the arbiter, the synchronizer, the latch, and the inertial delay. *IEEE Trans. on Comp.*, C-32(7):603–614, July 1983.

[3] Y. Brzozowsky. *Digital Networks.* Prentice-Hall, 1976.

[4] A. Cohn. Correctness properties of the VIPER block model: The second level. Technical Report 134, University of Cambridge, Computer Laboratory, May 1988.

[5] A. Cohn. A proof of correctness of the VIPER microprocessor: the first level. In P. Subrahmanyam G. Birtwistle, editor, *VLSI specification, verification, and synthesis*, pages 27–72. Workshop on hardware verification, Kluwer Academic Publishers, 1988.

[6] Th. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. In B. Buchberger, editor, *Proceedings EUROCAL'85*, pages 151–184, LNCS 203, 1985. Springer Verlag.

[7] Th. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[8] W. I. Fletcher. *An engineering approach to digital design.* Prentice-Hall, Englewood Cliffs, N.J., 1980.

[9] M. Fourman and E. M. Mayger. Formally based system design - Interactive hardware scheduling. In G. Musgrave and U. Lauther, editors, *Proceedings of the IFIP TC 10/WG 10.5 International Conference on VLSI, Munich, Aug. 16-18, 1989*, pages 101–112, 1989.

[10] Ganesh C. Gopalakrishnan, M. K. Srivas, and David R. Smith. From algebraic specifications to correct VLSI circuits. In D. Borrione, editor, *From HDL descriptions to guaranteed correct circuit designs*, pages 197–225. IFIP, North Holland, 1987.

[11] M. J. C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge, Computer Laboratory, July 1985.

[12] M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 73–128. Workshop on Hardware Verification, Kluwer Academic Publishers, 1988.

[13] T. G. Griffin. An environment for formal systems. Technical Report ECS-LFCS-87-34, Edinburgh Univ., Dept. of Comp. Sci., August 1987.

[14] F. K. Hanna and N. Daeche. Specification and verification using higher order logic: A case study. In G. M. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI design, Proc. of the 1985 Edinburgh conf. on VLSI*, pages 179–213. North-Holland, 1986.

[15] F. K. Hanna, N. Daeche, and M. Longley. VERITAS+:a specification language based on type theory. In *Proc. Conf. on Hardware Specification, Verification and Synthesis, Cornell University*, July 1989.

[16] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proceedings LICS'87*, pages 194–204, Ithaca, New York, June 1987.

[17] J. Herbert. Formal verification of basic memory devices. Technical Report 124, University of Cambridge, Computer Laboratory, February 1988.

[18] John Herbert. Temporal abstraction of digital design. In G. Milne, editor, *The fusion of hardware design and verification*, pages 1–25, University of Strathclyde, Glasgow, Scotland, July 1988. IFIP WG 10.2.

[19] W. A. Hunt, Jr. The mechanical verification of a microprocessor design. In D. Borrione, editor, *From HDL descriptions to guaranteed correct circuit designs*, pages 89–196. IFIP, North Holland, 1987.

[20] J. Joyce. Formal specification and verification of microprocessor systems. In S. Winter and H. Schumny, editors, *EUROMICRO'88*. North Holland, 1988.

[21] J. Lambek and P. J. Scott. *Introduction to higher order categorical logic*. Cambridge University Press, 1986.

[22] Zhaohui Luo. A higher order calculus and theory abstraction. Technical Report ECS-LFCS-88-57, Edinburgh Univ., Dept. of Comp. Sci., July 1988.

[23] L. R. Marino. *Principles of computer design*. Computer Science Press, Rockwell, 1986.

[24] I. Mason. Hoare's logic in the LF. Technical Report ECS-LFCS-87-32, Edinburgh Univ., Dept. of Comp. Sci., June 1987.

[25] D. May and D. Shepherd. Formal verification of the IMS T800 microprocessor. Internal report INMOS Limited, 1987.

[26] Thomas F. Melham. Abstraction mechanisms for hardware verification. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 267–292. Workshop on Hardware Verification, Kluwer Academic Publishers, 1988.

[27] B. Nordström. Martin-Löf's type theory as a programming logic. Technical Report 27, Chalmers University of Technology and University of Göteborg, September 1986.

[28] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's type theory. An introduction.* To be published by Oxford University Press, 1989.

[29] Randy Pollack. The theory of LEGO. Draft report, LFCS, Univ. of Edinburgh, October 1988.

[30] Brian Ritchie. *The Design and implementation of an interactive proof editor.* PhD thesis, Edinburgh Univ., Dept. of Comp. Sci., 1989.

[31] P. A. Subrahmanyam. Contextual constraints, temporal abstraction, and observational equivalence. In G. Milne, editor, *The fusion of hardware design and verification*, pages 156–182, University of Strathclyde, Glasgow, Scotland, July 1988. IFIP WG 10.2.

[32] P. A. Subrahmanyam. Towards a framework for dealing with system timing in very high level silicon compilers. In P. Subrahmanyam G. Birtwistle, editor, *VLSI specification, verification, and synthesis*, pages 159–215. Workshop on hardware verification, Kluwer Academic Publishers, 1988.

[33] P. Taylor and Z. Luo. Theories, mathematical structures, and strong sums. Preliminary notes, December 1988.

[34] S. H. Unger. *Asynchronous sequential switching circuits.* Wiley-Interscience, New York, 1969.

[35] D. W. Weise. *Formal multilevel hierarchical verification of synchronous MOS VLSI.* PhD thesis, Massachusetts Institute of Technology, 1986.

[36] S. Wendt. *Entwurf komplexer Schaltwerke.* Springer Verlag, Berlin, 1974.