Università
Ca'Foscari
Venezia

Dottorato di ricerca
in Informatica
Scuola di dottorato in Scienze e Tecnologie
Ciclo XXIV
(A.A. 2010 - 2011)

Hardware Execution of Constraint Handling Rules

SETTORE SCIENTIFICO DISCIPLINARE DI AFFERENZA: INF/01
Tesi di dottorato di Andrea Triossi, matricola 955599

Coordinatore del Dottorato	**Tutore del dottorando**
Prof. Antonio Salibra	**Prof. Salvatore Orlando**
	Tutore del dottorando
	Dr. Gaetano Maron

Università Ca' Foscari di Venezia

Dipartimento di Scienze Ambientali,
Informatica e Statistica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

Hardware Execution of Constraint Handling Rules

Andrea Triossi

Supervisor
Prof. Salvatore Orlando

Supervisor
Dr. Gaetano Maron

PhD Coordinator
Prof. Antonio Salibra

December, 2011

Author's e-mail: triossi@unive.it

Author's address:

Dipartimento di Scienze Ambientali,
Informatica e Statistica
Università Ca' Foscari di Venezia
Via Torino, 155
30172 Venezia Mestre – Italia
tel. +39 041 2348411
fax. +39 041 2348419
web: `http://www.dsi.unive.it`

“I’m sure I’ll take you with pleasure!” the Queen said.
“Two pence a week, and jam every other day.”

Alice couldn’t help laughing, as she said,
“I don’t want you to hire ME — and I don’t care for jam.”

“It’s very good jam,” said the Queen.

“Well, I don’t want any TO-DAY, at any rate.”

“You couldn’t have it if you DID want it,” the Queen said.
“The rule is, jam to-morrow and jam yesterday — but never jam to-day.”

“It MUST come sometimes to ‘jam to-day’,” Alice objected.

“No, it can’t,” said the Queen.
“It’s jam every OTHER day: to-day isn’t any OTHER day, you know.”

“I don’t understand you,” said Alice. “It’s dreadfully confusing!”

Lewis Carroll
“Through the Looking Glass and What Alice Found There”

Abstract

In this thesis we want to investigate the compiling of the well-established language Constraint Handling Rule (CHR) to a low level hardware description language (HDL). The benefit introduced by a CHR-based hardware synthesis is twofold: it increases the abstraction level of the common synthesis work-flow and it can give significant speed up to the execution of a CHR program in terms of computational time.

We want to propose a method that sets CHR as a starting point for a hardware description. The developed hardware will be able to turn all the intrinsic concurrency of the language into parallelism. The rules application is mainly achieved by a custom executor that handles constraints according to the best degree of parallelism the implemented CHR specification can offer.

Afterwards we want to integrate the generated hardware code, deployed in a Field Programmable Gate Array (FPGA), within the traditional software execution model of CHR. The result will be a prototype system consisting of a CHR execution engine composed of a general purpose processor coupled with a specialized hardware accelerator. The former will execute a CHR specification while the latter will unburden the processor by executing in parallel the most computational intensive rules.

Finally the performance of the proposed system architecture will be validated by time efficiency measures.

Acknowledgments

I wish to thank my supervisor Salvatore Orlando for having made my PhD a productive and stimulating experience. He let me choose my research topic in complete freedom and then he has steadily pointed me the way. I am really grateful to him for his practical advice and invaluable support.

I am thankful to Gaetano Maron who gave me the chance to keep on working at INFN. Somebody could say that it is a way to extend an agony, but till now it has been just a splendid opportunity for working in a challenging, dynamic, professional environment.

I gratefully acknowledge Thom Frühwirth who had patience to guide a physicist through the CHR world. I admire his expertise on every single aspect of CHR, even when it involves applications in varied and different research areas. His devotion towards all the projects he follows lavished enthusiasm on me.

Thanks to the co-authors Alessandra Raffaetà, Frank Raiser for their important and essential contribution. I wish also to acknowledge Paolo Pilozzi for all the time he spent giving me fundamental suggestions for my research topic.

All the PhD students of the computer science department of Ca'Foscari, and in particular Stefano, have supported me during last three years always answering to my naive questions.

I would like to thank all the colleagues of Gamma laboratory in Padova and all the people who gravitate around the coffee machine: Dino, Sandro, Roberto, Gabriele, Damiano, Francesco, Francesco, Diego, Daniele, Marco. A warm thanks to Marco Bellato *the boss of it all*. Besides a lot of very useful "engineering stuff" he taught me the fairness that such particular work requires.

To conclude I am afraid, but I must admit that I cannot acknowledge Chiara better than she did me. So "I won't say anything at all, and I'll offer *her* a beer!"

Contents

List of Figures

List of Tables

1 Introduction

It is well known that, while high level languages (HLLs) simplify the live of the programmers easing program encoding, low level implementations are more efficient. Nevertheless in particular settings, thanks to powerful compilation techniques, HLLs can produce code comparable in efficiency to their low level counterparts [22]. With the work presented in this thesis we want to take a bigger leap, from declarative programming to very low level language mapping high level algorithmic code directly to a hardware representation.

We identify the HLL, that can fulfill the hardware description requirements, in Constraint Handling Rules (CHR) [46]. CHR is a committed-choice constraint logic programming language first developed for writing constraint solver, and nowadays well-known as general-purpose language. Its plain and clear semantics makes it suitable for concurrent computation, since programs for standard operational semantics can be easily interpreted in a parallel computation model [45]. Furthermore CHR does not allow backtracking search but it rather employs guards that are used to commit to a single possibility without trying the other ones. When a goal cannot be rewritten, the process solving this goal does not fail but it is blocked until, possibly, other processes will add the constraints that are necessary to entail the guard of an applicable clause. Hardware description needs straightforward processes that cannot be undone as well, and moreover it benefits from guards as they can be used for synchronization among processes solving different goals. In our work we want to fully exploit these features, highlighting the relations with the parallel characteristics of the target gate-level hardware.

From the point of view of the hardware compilation the search of a shortcut between HLLs and hardware description languages (HDLs) represents a significant improvement in terms of abstraction level. Indeed, the ability of integrating an ever greater number of transistors within a chip and the consequent complexity growth of the features that can be implemented in hardware has demanded for new languages and tools suitable for describing the different natures of the hardware devices. Integrated devices may well contain several processors, memory blocks or accelerating hardware units for dedicated functions that are more related to software architectures than low level hardware representations. Hence design automation technology is now seen as the major technical barrier to progress: tools adopted to describe complex systems rise an explicit demand for design representation languages at a higher abstraction level than those currently adopted by hardware system engineering. According to this research direction, one of the goals of this thesis is to synthesize hardware starting from a language at a level higher than the ones of the commonly used behavioral HDLs. Such enhancement should let programmers easily focus on system behavior rather than on low level implementation details. The design procedure identified in [61] can be applied to a declarative paradigm rather than traditional imperative languages, inheriting all the well-known benefits for the programmer.

Moreover, we aim to investigate whether the same hardware synthesis methodology can be exploited to statically generate a specialized hardware accelerator, which couples a commodity CPU along with a modified run-time support for executing a general purpose declarative program. In such a way we achieve a close synergy between high level software and hardware because the former facilitates the synthesis of the latter and the latter clears the way for a fast execution of the former. Hardware/software co-design has nowadays a long tradition in system design and is rightfully considered a practical design task [113]. Since early '90s performance and cost analysis have suggested to deploy some of the program's functions in specialized hardware, namely Application Specific Integrated Circuit (ASIC), while the remaining part of the program is implemented in software and executed on the CPU [52]. In the following years co-design evolved and many research fields come to light, for example: partitioning, co-simulation, co-synthesis or co-verification. Gradually the proposed methodologies moved from addressing hardware and software as distinct entities to consider them at a higher level of abstraction that encompasses both of them [86].

Research contributions

In response to the aforementioned motivations we first develop a completely general framework that allows one to synthesize reconfigurable hardware that can be easily employed in a wide range of applications since a small modification to the high level code affects a huge portion of low level HDL code (resulting in a remarkable time saving for the programmer). Moreover the generated hardware code is fully compliant with the traditional standard ones, hence it can be easily integrated in the existing hardware project. Once hardware can be directly compiled from a high level language, we want to implement a coupled system constituted by a traditional general purpose CPU and a hardware accelerator deployed on a Field Programmable Gate Array (FPGA). Thus such unique system will be compiled from a single high level language specification through a double compilation path resulting in an efficient execution engine.

Specifically the thesis provides the following contributions:

- The development of a novel technique to synthesize behavioral hardware components starting from a subset of CHR;

- An implementation of an efficient and optimized parallel execution model of CHR by means of hardware blocks; and
- The development of a custom reconfigurable hardware that significantly speeds up the execution time of a CHR program

The proposed method for high level hardware synthesis and some partial results of such technique already appeared in [106]. While the general outline of the hardware accelerator that aids the CHR computation was presented in [105].

Plan of the thesis

This thesis is organized as follows. The next two Chapters provide the needed background regarding the software and hardware frameworks exploited for pursuing the goals of the thesis. Chapter 2 starts with a brief introduction of the paradigms related to CHR in order to introduce the reader to a better comprehension of the rest of the Chapter that focuses on the CHR language and its property. In Chapter 3 we will give a review of the reconfigurable computing from the hardware to the high level language commonly adopted for its programming.

The following Chapters deal with the hardware emulation of CHR: they span the compilation from the hardware synthesis to the employment of the generated hardware as a CHR accelerator. The central thread of this section is the performance optimizations that will be achieved through the high degree of parallelization that hardware can supply. In Chapter 4 we focus on the technique adopted for generating hardware block from CHR rules. While in Chapter 5 we show how to efficiently accommodate in hardware parallel rules execution. Finally Chapter 6 illustrates how the reconfigurable hardware can fit into a complete computing architecture. Here a detailed study of the trade-offs of such technique is presented. Beyond the running example that drives the reader through the description of the parallelism between CHR and hardware, a complete practical example of implementation is provided within each Chapter. Classical algorithms usually adopted for showing the expressiveness of CHR in multiset transformation or constraint solving, are chosen as case studies.

Related works are mainly arranged across the thesis into three Sections concerning: parallel execution models of CHR 2.2.3; high level languages adopted for the hardware synthesis 3.3.2; and hardware accelerators 6.1.

I

Background

2 Constraint Handling Rules

Constraint handling rules (CHR) [46] is a logical concurrent committed-choice rule-based language consisting of guarded rules that transforms constraints until a fixed point is reached. CHR was originally designed for the specific purpose of adding user-defined constraints solvers to a host language, and nowadays is evolved in a general-purpose language with many application domains [47, 2, 6, 17, 114, 26, 104]. As we will see later in the second part of this thesis the framework for hardware synthesis we will propose is strongly related with CHR since it is based on its clear syntax and semantics. An introduction to such programming language is therefore mandatory for a proper comprehension of the synthesis mechanism reported in Chapter 4.

In the first Section of this Chapter the programming paradigms related to CHR are informally introduced. In Section 2.2 we will focus on CHR and its operational semantics, starting with some simple examples of CHR computation and then giving a detailed description of the operational semantics. Finally we will discuss the concurrency property of CHR, which will turn out to be essential during the process of hardware execution of a CHR specification.

2.1 Declarative Programming

In contrast with imperative programming the declarative paradigm focuses on *what* has to be computed rather than *how* to compute it [73]. In a declarative framework the program is a theory and the computation is a deduction from the theory in some defined logics. For example first order logics for the most of logic programming languages or λ-calculus for functional ones. If the programs are just theory the *control* part should be automatically supplied (at least in the ideal case) by the system. In general we cannot say that this is totally desirable: even in the most radical declarative programming environment the programmer should be left free to ensure that the upper bounds of the time and space complexities are not exceeded. The problem of the degree of *control* grows stronger in logic programming because of the explicit non-determinism provided by logic languages, but in principle there is no need to have a system fully bereft of control facilities although it is advisable to reduce them to a minimum.

The practical advantages of declarative languages over other formalisms can be summarized as follows:

- Declarative program are usually more clean and succinct. Hence they appear more readable, easy to reuse/adapt/maintain or in one word self-explanatory.
- Simple semantics compared with the semantics of the majority of imperative languages. The benefits introduced by a plane semantics are not only related to a more straightforward theoretical abstraction, since they can lead to an easier program analysis and hence program optimization, verification and debugging.
- Leaving undefined *how* the program should behave in general entails more room for parallelism in the declarative paradigm. Indeed implicit parallelism is present in the most declarative implementations, thus reducing the need for concurrent constructs and leaving more freedom to the compiler to match the underlying hardware architecture.
- Finally all the above mentioned advantages can lead to a reduced development time for software programmers letting them concentrate more on the algorithm to be implemented.

Until now we defined declarative paradigm as a general concept, but to treat the argument in a more detailed way we should define the formalism used to specify programs. To categorize the wide variety of programming paradigms is not an easy task, but we list the most well-known ones:

- Functional Programming
- Logic Programming
- Constraint Programming
- Rule-based Programming

Constraint Handling Rules (CHR), as we will discuss in Section 2.2, encompasses the last three paradigms.

2.1.1 Functional Programming

Functional programming is a paradigm base on the mathematical framework of the λ-calculus originated in the '30s to formally describe functions and their application and recursion [75]. Its key concept is the functions evaluation avoiding states and mutable data: the mathematical functions expressed by functional programming are supposed to be lacking in *side effect*. This means that, in order to avoid states modification, functions cannot modify either global variables or one of their arguments or everything that should make the program's behavior depending on the execution history. Hence in functional programming the result of a function depends only on its inputs. There are several reasons to want to avoid *side effect.* For example when a result of a function is not used the entire expression can be removed. Moreover if two functions have no dependency their execution order can be swapped or even parallelized. And finally since the result of a function depends only on its inputs it is possible to use cache optimizations.

As examples of functional languages we can cite Haskell [78], Clean [20] or Scheme [38]. The main difference among them is the evaluation strategy: it can be *strict (eager)* or *non-strict (lazy).* In the first case function evaluation is done only after the evaluation of all its arguments, while in the second one the argument evaluation is not done unless strictly necessary to the final result. Another important difference is represented by how they treat I/O without introducing *side effects.* Indeed monads and uniqueness are example of technique to avoid *side-effect* in I/O as well. Monads, employed in Haskell, are abstract data type constructors that encapsulate program logic, while uniqueness, used in Clean, is intended to guarantee that an object is used with at most a single reference to it.

2.1.2 Logic Programming

The main idea behind logic programming is to use first order logic, or at least a subset of it, as a programming language. The program is considered as a logical statement while the execution becomes a form of theorem proving. A complete introduction of logic programming can be found in [72] or in the survey [71]. Here we just report some concepts that will ease the comprehension of the rest of the Chapter.

The basic elements of logic programming are a set of *variables*, a set of *functions* and a set of *predicates.* The former are indicated with capital letter words while for the latter two the *arity* is defined as their number of arguments they have and it is usually preceded by a / (e.g. f/n means that the function or predicate f has arity n). Zero-arity denotes constants. Logical *variables* differ from the classical ones used in imperative programming since they are really unknown elements: they do not indicate any value, but they may be even used if they do not refer to any value.

A *term* is a variable or a *compound term* that is inductively defined as $f(t_1, ..., t_n)$ where f is a function of *arity* n and t_i are *terms.* A *ground term* is a *term* without any *variables.* An *atom* is a *predicate* applied to a sequence of *terms.* We call *literals* the negation of an *atom* and *expression* both *terms* and *atoms.*

A *substitution* $\sigma = (X_1 = t_1 \wedge \ldots \wedge X_n = t_n)$ is defined as the conjunctions of the bindings $X_i = t_i$ with $X_i \neq t_i$. *Unification* of two *expressions* E_1 and E_2 is the *substitution* σ for which $\sigma(E_1) \equiv \sigma(E_2)$ where $\equiv$ is the *syntactical equivalence.*

A logic program consists of a collection of rules, called *clauses*, of the form:

$$H\,\text{:-}L_1, \ldots, L_n\,. \tag{2.1}$$

Where H is an *atom*, called *head* of the *clause*, and L_i for $i \in \{1 \dots n\}$ are *literals* that form the *body*. The comma denotes the logical conjunction that depends on the semantics of the program. If the *body* is empty the *clause* is called a *fact*.

The inference method used for evaluating a logic program is called Linear resolution with a Selection function applied to Definite clauses (SLD resolution). The abstract semantics introduced by SLD resolution adopts a backwards chaining inference method that, starting from a conjunction of *literals* called *goals* or *query*, proceeds until a *fact* or a built-in knowledge. Hence, in order to resolve the *goal* h, we should check if it unifies with the head of a *clause* (for example the one in 2.1), and then it is sufficient to resolve the *literals* L_i. If the whole *query* can be resolved the program succeeds and the *variables* generated by all the bindings are the solution.

The abstract semantics of SLD resolution is clearly non-deterministic since it does not specify which should be the next *"active" goal* in a conjunction of *literals* nor in which order the *clauses* are evaluated. Indeed a *goal* can unify with many *heads* giving rise to many alternative computations that correspond to the branch of a search tree. If from the root node (the initial *goal*) we can reach a successful leaf (a leaf in which the conjunction of *goals* is empty) we say that the goal succeeds otherwise, if there are no possible paths to a successful leaf, the *goal* fails.

SLD resolution operational semantics want to prune the non-determinism introduced by the abstract semantics and usually adopts a left-to-right selection methods that chooses the next *literal* to evaluate. Moreover it tries to execute the *clauses* in a top-down order which means that if a *clause* leads to a failure it rollbacks the computation (unbinding all the unified *variables* until the previous node) and keeps the computation going, trying the next *clause*. When a subtree exploration results in a failure, the movement of coming back to the former node (choice-point) is called *backtracking*.

Prolog

Prolog was the first practical implementation of logic programming: its born date back to the 70's but it remains the most widely used general purpose logic programming language [28]. Prolog implements an extension of SLD resolution operational semantics that provides negation called SLD-NF (negation-as-failure). In such logic the negation of a *goal* G succeeds when G fails. Other features introduced by Prolog are: disjunction, lists, arithmetics, built-in *predicates*. Disjunction in the *body* of a *clause* simply adds an additional choice-point with two distinct branches for the two disjoint *literals*. Lists are *terms* that are able to aggregate a (possibly empty) ordered collection of terms. Since Prolog supports integer and floating point data types it provides for arithmetic operations. The built-in predicate `is/2` is employed for evaluating the arithmetic expression and unifying the result to a *variable*. Finally, in addition to user defined *predicates* and unification, Prolog provides for a number of predefined built-in *predicates* that can have a logical or non-logical meaning. Examples of the first category are the *predicates* `true`/0 and `false`/0 that, respectively, always and never succeeds, while input/output *predicates*, like `writeln`/1, belong to the second category.

2.1.3 Constraint Programming

Constraint programming differs from the other paradigms in the sense that relations among variables are described through constraints. Constraints solvers come usually embedded in other programming languages or in separate libraries. In this Section we are interested in constraints embedded in logic programming.

Constraint Solver

We define *primitive constraints*, c/n, as a *predicate* that for every argument position has either a *variable* or a value from a value set V_i. Such definition recalls the one of *atom* in logic programming.

A *constraint* has the form of $c_1 \land \ldots \land c_n$ where c_i are *primitive constraints.* A *constraint domain* $\mathcal{D}$ consists of a restriction in the set of possible *predicates* on which the *primitive constraints* are defined. Given a logical theory $\mathcal{T}$ we can state whether a constraint holds. The notation $\mathcal{D} \models c$ denotes that the constraint c holds under $\mathcal{T}$ in domain $\mathcal{D}$.

The variable substitution that maps the *variables* of a *c*onstraint C into values belonging to a domain $\mathcal{D}$ is called *valuation.* A valuation ϕ is a *solution* for C if $C\phi$ holds in $\mathcal{D}$ ($\mathcal{D} \models C\phi$). A *constraint* is called *satisfiable* if it has a *solution* and two *constraints* are *equivalent* if and only if they have the same *solution.* Finally a *constraint solver* is an algorithm that determines the satisfiability of a *constraint.*

Constraint Logic Programming

The combination of the efficiency of *constraint solving* and the declarativity of Logic Programming results in Constraint Logic Programming. In such a context the *body* of a *clause* may have a mix of traditional *literals* and *constraints.*

A new operational semantics rises from logic programming one with an extension due to the integration of a *constraint solver.* We define the *constraint store* as the conjunction of all the *constraints* encountered during computation that starts from an empty *constraint store* $\mathcal{S}$. Each time that a *constraint* c is added to the store the new *constraint store* becomes $\mathcal{S}' = \mathcal{S} \land c$. After insertion the *constraint solver* is charged to rewrite the *store* to a *solved form* (a constraint that appears clear whether it is satisfiable). Here the logic programming resolution mechanism is influenced since if the solved form is unsatisfiable it fails otherwise it resumes. Besides the incrementation of the *store* another operation is provided. Given a *constraint* we can also check if the whole *constraint store* entails it and in a similar manner we can state if this operation fails or resumes the resolution. The former operation is called *tell constraints* while the second one is called *ask constraints*.

In [57] a wide variety of constraint programming languages is presented. Basically they belong to two different category depending if they are standalone or embedded in some host language like Prolog, Java or C. The first constraint programming systems were difficult to modify (add new data types or constraints) but recent systems are fully customizable while maintaining a good degree of efficiency [85].

Concurrent Constraint Logic Programming

Concurrent Constraint logic Programming (CCP) appears for the first time in the '90s in [87] but the main idea was already introduced a few years before in [76]. As the name suggests the aim of such programming paradigm is the evaluation of the *goals* by means of concurrent processes. The most significant difference with CLP is how the interpreter behaves in case of applicability of more than one *clause*: instead of trying every possible *clause* until an answer is found it applies a series of clauses until failure or success without *backtracking*. In order to condition the execution of a clause CCP implements *guards*: a further check for the applicability of a *clause* to be done after the head matching. Concurrent processes are generated by the application of the *clauses* that can communicate via a shared global *constraint store* and *guards* can be used for synchronization purposes. Indeed, besides conditions among *variables*, *ask* and *tell* checks can be also implemented in the *guards*.

2.1.4 Rule-based Programming

Rule-based Programming commonly refers to a wide family of programming languages that derive from the *production rule systems* initially used for artificial intelligence applications whereas nowadays they are usually adopted for business rules.

A production system basically consists of a *working memory* and several *condition-action rules*. The first is a collection of *facts* that are runtime instance of a *template*. The *condition-action rules* are formed by a *left hand side* (LHS) that is a conjunction of *condition elements* used to specify the circumstances of application of a rule, and by a *right hand side* (RHS) that is a conjunction of *actions*. The *condition elements* are divided in positive and negative depending if the *facts* they refer must belong to the WM or not. The possible *actions* that a rule can perform are combinations of addition and removal of a *fact* to and from the WM.

One of the first *production rule system* is OPS that appears in the late '70s [43]. The main field of application of such language was the *expert systems*: an artificial intelligent system able to provide answers to questions that usually need an expert intervention like the choice of a product from the requirement of the costumer. In the following years production systems evolve from the more efficient CLIPS [50] developed by NASA through the Java compliant Jess [58] and finally to the modern business rule management systems like JBoss Enterprise or PegaRULES that are complete software platform integrating the rule engine with graphical user interface for rule editing.

2.2 Constraint Handling Rules

Constraint Handling Rules is a multi-headed guarded constraint logic programming language [46]. It employs two kinds of constraints: built-in constraints, which are predefined by the host language, and CHR constraints, which are user-defined by program rules. Each CHR constraint is taken by the predicate symbols and is denoted by a *functor* (or name) and a *arity* (number of the arguments). Null-arity built-in constraints are `true` (the empty constraint) and `false` (any inconsistent constraint). A CHR program is composed of a finite set of rules reasoning on constraints. We can distinguish two kinds of rules:

$$\text{Simplification:} \quad Name \ @ \ H \Leftrightarrow G \mid B \tag{2.2}$$

$$\text{Propagation:} \quad Name \ @ \ H \Rightarrow G \mid B \tag{2.3}$$

Where $Name$ is an optional unique identifier of the rule, H (*head*) is a non-empty conjunction of CHR constraints, G (*guard*) is an optional conjunction of built-in constraints, and B (*body*) is the goal, a conjunction of built-in and CHR constraints. These rules logically relate head and body provided that the guard is true. Simplification rules mean that the head is true if and only if the body is true and propagation rules mean that the body is true if the head is true. Rules are applied to an initial conjunction of constraints (*query*) until no more changes are possible. The intermediate goals of a computation are stored in the so called *constraint store*. During the computation if a simplification rule fires the head constraints are removed from the store and they are replaced by the body constraints. If the firing rule is a propagation rule the body constraints are added to the store keeping the head constraints. A third rule called simpagation permits to perform both a simplification and propagation rule:

$$\text{Simpagation:} \quad Name \ @ \ H_1 \backslash H_2 \Leftrightarrow G \mid B \tag{2.4}$$

This rule means that the first part of the head (H_1) is kept while the second (H_2) is removed from the constraint store. It is logically equivalent to the other two rules depending on which part of the head is empty. One rule is in *head normal form* when all the arguments in the head of the rule are variables and variables never occur more than once (all equality guards implicitly present in the head are written explicitly in the guard).

Example 2.2.1. The following CHR program computes the greatest common divisor (gcd) between two integers using the Euclid's algorithm.

```
R0 @   gcd(N) <=> N = 0 | true.
R1 @   gcd(N) \ gcd(M) <=> M>=N | Z is M-N, gcd(Z).
```
(2.5)

Rule `R0` is a simplification rule written in its normal form. It states that the constraints `gcd` with the argument equal to zero can be removed from the store. Rule `R1` is a simpagation rule since there is a kept and a removed part of the head. It is written in its normal form as well and it states that if two constraints `gcd(N)` and `gcd(M)` are present, the latter can be replaced with `gcd(M-N)` if `M>=N`.

An example of query for the gcd program can be for instance:
`gcd(12), gcd(27), gcd(9), gcd(24), gcd(6).`
It will result in `gcd(3)` since repeated applications of rule `R1` to couple of constraints followed by `R0` exclude all the possible candidate `gcd` except the generated `gcd(3)`.

Example 2.2.2. Another very simple example of CHR program is the family relationship property of being mother of somebody. With the constraint `mother(X,Y)` we want to state that `Y` is the mother of `X`. By the next rules we want to express the uniqueness of the mother and the how deduce the grandmother relationship from the mother one.

```
dM @   mother(X,Y) \ mother(X,Z) <=> Y=Z.
GM @   mother(X,Y), mother(Y,Z) ==> grandmother(X,Z).
```
(2.6)

Rule `dM` uses the built-in syntactic equality `=` to make sure that both variables (`Y` and `Z`) have the same value. That means that occurrences of one variable are replaced by the other. One `mother` constraint is discarded by the rule since, once the variables are made equal, it represents a duplicate of the other. For instance a query `mother(tom,mary), mother(tom,mary).` will just result in the removing of one duplicate constraint since the introduction of `mary=mary` will be simplified because lead to `true`. On the contrary a query like `mother(tom,mary), mother(tom,lisa).` will result in `false` since the equality `mary=lisa` fails. The rule `GM` instead propagates the grandmother relationship from the mother one. A simple query:
`mother(tom,mary), mother(mary,lisa).`
will generate `grandmother(tom,lisa)` without removing any constraints.

2.2.1 Semantics

The CHR semantics comes in different flavors that can be divided into two main families: the logical and the operational ones. The former represents the formal basis of CHR while the latter states the behavior of actual implementations. We will focus our attention on the operational semantics that comprises the very abstract, the standard (or abstract or theoretical) and the refined semantics. The following sections describe these three semantics denoted respectively as ω_a, ω_t and ω_r.

Very Abstract Operational Semantics ω_a

The very abstract operational semantics [48] is a non-deterministic state transition system where the execution proceeds by exhaustively applying the transition rules.

Figure 2.1: Very abstract operational semantics transition system

$$\begin{array}{ll} \textit{Apply}: & \langle H_1 \wedge H_2 \wedge G \rangle \longmapsto_{\textit{apply}} \langle H_1 \wedge C \wedge B \wedge G \rangle \\ & \text{where it must exist in the program an instance} \\ & \text{with new local variables of rule } r \text{ of the form} \\ & r \ @ \ H_1 \backslash H_2 \Leftrightarrow C \mid B \\ & \text{and } \mathcal{D}_b \models G \longmapsto \bar{\exists} C, \end{array} \tag{2.7}$$

The relation between an execution state and the following one is defined by only one possible transition rule called *Apply*. The states are represented by just conjunction of built-in and CHR constraints and the initial state is given by the goal.

The transition between states corresponds to the application of the rule presented in Figure 2.1. H_1, H_2, C, B and G are, possibly empty, conjunction of constraints. An instance with fresh variables of r is applicable if its head constraints are present in the state and the guard C is logically implied by the built-in constraints in G, that corresponds to state that the condition $\mathcal{D}_b \models G \longmapsto \bar{\exists} C$ holds. As consequence of the rule application the constraints in H_1 are kept while the ones in H_2 are removed from the state. The guard and the body, C and G, are also added to the resulting state.

The ω_a transition system is definitely non-deterministic because it does not establish which rule should be applied to a given state. If more then one rule can fires one is chosen in a non-deterministic way and the choice cannot be undone since CHR is a committed choice language.

Example 2.2.3. Considering the CHR program introduced in Example 2.2.1 we report in Table 2.1 the derivation under ω_a for the query $\{gcd(6), gcd(9)\}$.

Table 2.1: Example of derivation in very abstract semantics

Initial state	$\langle gcd(6) \wedge gcd(9) \rangle$
ApplyR1	$\langle gcd(6) \wedge gcd(3) \rangle$
ApplyR1	$\langle gcd(3) \wedge gcd(3) \rangle$
ApplyR1	$\langle gcd(0) \wedge gcd(3) \rangle$
ApplyR0	$\langle gcd(3) \rangle$

Standard Operational Semantics ω_t

The standard operational semantics [1] is a state transition system that, unlike the very abstract semantics, takes more care about termination. Indeed in ω_a the same propagation rule can fire an unlimited number of time because additional built-in constraint cannot invalidate a condition that holds (without resulting in a failure). Moreover, since in a failed state any rule satisfies the applicability condition, a failed state can only lead to an other failed state resulting in a not terminating execution.

To overcome these issues the standard semantics considers a three-rules-transition system: *Solve*, *Introduce* and *Apply*. The first rule solves a built-in constraint from the goal, the second inserts a goal constraint in the store and the third fires an instance of a CHR rule. To prevent trivial non-termination, propagation rules cannot be applied twice on the same constraints and the final state is reached when either any transition rule is no more applicable or a built-in constraint is not satisfied. To formally describe ω_t we recall the version presented in [36] that is completely equivalent to the previously formulated one.

An *execution state* σ is a tuple $\langle G, S, B, T\rangle_n$ where G is the multiset of constraints (called *goal*) that has to be rewritten, S and B are multisets of CHR and built-in constraints respectively that form the *constraint store*, T is the *propagation history* (used to ensure termination) and n is the next free *identifier* to be assigned to a constraint. Each constraint c in the store S has an unique integer *identifier* i and is denoted as $c\#i$. The *identifier* is needed to distinguish among copies of the same constraint. The *propagation history* is a sequence of the constraints that fired a rule and the rule itself. Those data are recorded with the aim of avoid a trivial non-termination in presence of propagation rules. Indeed a propagation rule must not fire twice consecutively on the same set of constraints. If G is a multiset of constraints the transition begins from the *initial execution state* $\langle G, \emptyset, true, \emptyset\rangle_n$.

The three rules that allow moving among states are formally described in Table 2.2. In the last rule we have used the notation $\circ$ for the operator of sequential concatenation and the function *chr* and *id* as projectors of the pair $c\#i$ (i.e. $chr(c\#i) = c$ and $id(c\#i) = i$). The application of the transition requires a matching substitution θ and the validity of the guard g. Finally the *propagation history* T is incremented by the *identifier* of the constraints H_1, H_2 and by the rule r.

The application order of the transitions is non-deterministic. Usually starting from an initial state many different transitions are applicable and hence many different states are reachable. A positive termination is achieved when we cannot apply any other transition rule, instead when the constraint solver can prove $\mathcal{D}_b \models \neg\bar{\exists}_\emptyset B$ we are in presence of a failed derivation. $\mathcal{D}_b$ denotes the constraint domain of the built-in constraints.

Example 2.2.4. Like in Example 2.2.3, in Table 2.3 is reported abstract semantics derivation of the gcd program for the query $\{gcd(6), gcd(9)\}$. The computation successfully terminates when no more rule can be applied. The only chr constraint remained in the store is the resulting constraint $gcd(3)$.

Table 2.2: Standard operational semantics transition system

$$\mathit{Solve}: \quad \langle \{b\} \uplus G, S, B, T\rangle_n \longmapsto_{\mathit{solve}} \langle G, S, b \wedge B, T\rangle_n \tag{2.8}$$

where b is a built-in constraint

$$\mathit{Introduce}: \quad \langle \{c\} \uplus G, S, B, T\rangle_n \longmapsto_{\mathit{introduce}} \langle G, \{c\#n\} \uplus S, B, T\rangle_{n+1} \tag{2.9}$$

where c is a CHR constraint

$$\mathit{Apply}: \quad \langle G, H_1 \uplus H_2 \uplus S, B, T\rangle_n \longmapsto_{\mathit{apply}} \langle C \uplus G, H_1 \uplus S, \theta \wedge B, T'\rangle_n \tag{2.10}$$

where it must exist in the program a rule r of the form

$r \text{ @ } H_1' \backslash H_2' \Leftrightarrow g \mid C$

and a substitution θ such that $chr(H_1) = \theta(H_1')$,

$chr(H_2) = \theta(H_2')$, $\mathcal{D}_b \models B \longmapsto \bar{\exists}_B(\theta \wedge g)$,

and $T' = T \circ \{id(H_1) \circ id(H_2) \circ [r]\}$

Refined Operational Semantics ω_r

The refined operational semantics [60] [36] removes much of the non-determinism of the standard semantics. It uses a stack of constraints that is built up taking one by one the goal constraints and the constraints introduced by the application of a rule. The constraint at the top of the stack is called active and in turn each active constraint attempts to match the head of a rule together with the constraints present in the store. The main difference with respect to the standard operational semantics relies on the fact that transition rules like *Solve* and *Introduce* cannot be applied on any constraint in G. Thus CHR constraints are considered like a sort of procedure calls because, when a new constraint is picked up from the stack, it triggers the matching mechanism with all the program's rules in order until the execution or the deletion of such constraint from the stack. In refined semantics the constraints insertion of the rule's body is left-to-right and the rules are evaluates in a top-to-bottom order. Such determinism makes the refined semantics an instance of the standard one and hence any correct program for standard is correct for refined as well. Unfortunately the opposite implication is valid only in presence of *confluence.* With confluence we mean the property of producing the same results independently of the rules application order (see Section 2.2.2 for further details).

The refined operational semantics can be formally described as a transition system acting on states that have the form $\langle A, S, B, T\rangle_n$ where A is the execution stack

Table 2.3: Example of derivation in standard semantics

Initial state	$\langle\{gcd(6), gcd(9)\}, \emptyset, true, \emptyset\rangle_0$
Introduce	$\langle\{gcd(9)\}, \{gcd(6)\#0\}, true, \emptyset\rangle_1$
Introduce	$\langle\emptyset, \{gcd(6)\#0, gcd(9)\#1\}, true, \emptyset\rangle_2$
Apply $_{R1}$	$\langle\{Z\ is\ 9-6, gcd(Z)\}, \{gcd(6)\#0\},$ $gcd(M) = gcd(9) \wedge gcd(N) = gcd(6) \wedge M \geq N,$ $\{0, 1, [R1]\}\rangle_2$
Solve	$\langle\{gcd(3)\}, \{gcd(6)\#0\}, true, \{0, 1, [R1]\}\rangle_2$
Introduce	$\langle\emptyset, \{gcd(6)\#0, gcd(3)\#2\}, true, \{0, 1, [R1]\}\rangle_3$
Apply $_{R1}$	$\langle\{Z\ is\ 6-3, gcd(Z)\}, \{gcd(3)\#2\},$ $gcd(M) = gcd(6) \wedge gcd(N) = gcd(3) \wedge M \geq N,$ $\{0, 1, [R1], 2, 0, [R1]\}\rangle_3$
Solve	$\langle\{gcd(3)\}, \{gcd(3)\#2\}, true, \{0, 1, [R1], 2, 0, [R1]\}\rangle_3$
Introduce	$\langle\emptyset, \{gcd(3)\#2, gcd(3)\#3\}, true, \{0, 1, [R1], 2, 0, [R1]\}\rangle_4$
Apply $_{R1}$	$\langle\{Z\ is\ 3-3, gcd(Z)\}, \{gcd(3)\#2\},$ $gcd(M) = gcd(9) \wedge gcd(N) = gcd(6) \wedge M \geq N,$ $\{0, 1, [R1], 2, 0, [R1], 2, 3, [R1]\}\rangle_4$
Solve	$\langle\{gcd(0)\}, \{gcd(3)\#2\}, true, \{0, 1, [R1], 2, 0, [R1], 2, 3, [R1]\}\rangle_4$
Introduce	$\langle\emptyset, \{gcd(3)\#2, gcd(0)\#4\}, true, \{0, 1, [R1], 2, 0, [R1], 2, 3, [R1]\}\rangle_5$
Apply $_{R0}$	$\langle\emptyset, \{gcd(3)\#2\}, gcd(N) = gcd(0),$ $\{0, 1, [R1], 2, 0, [R1], 2, 3, [R1], 4, [R2]\}\rangle_5$

of constraints S and B are multisets of CHR and built-in constraints respectively, T is the propagation history (used to ensure termination) and n is the next free identifier to be assigned to a constraint. Basically S,B and T, have the same meaning than in the standard semantics while A is a sequence of identified constraints $c\#i$ and *occurrenced* identified constraints. An *occurrenced* identified constraint $c\#i : j$ is an identified constraint that can only match with occurrence j of the constraint c. It is worth noting that in refined semantics the same identified constraint can be present at the same time both in the execution stack A and in the store S.

The initial state is $\langle G, \emptyset, \top, \emptyset\rangle_1$, where G is the goal, and the possible final states can be either $\langle G, S, \bot, T\rangle_n$, if the computation has failed, or a given state in which no more transition rules are applicable. The transition system is composed of seven rules as reported in Table 2.4.

As in the standard semantics, transition *Solve* is used to increment the built-in store: when it is applied a built-in constraint b is moved from the goal to the store. Since the introduction of a new built-in can change the applicability of a guard, *Solve*

Table 2.4: Refined operational semantics transition system

Solve :
$$\langle [b|A], S_0 \uplus S_1, B, T\rangle_n \longmapsto \langle S_1 + +A, S_0 \uplus S_1, b \wedge B, T\rangle_n \tag{2.11}$$
where b is a built-in constraint

Activate :
$$\langle [c|A], S, B, T\rangle_n \longmapsto \langle [c\#n : 1|A], \{c\#n\} \uplus S, B, T\rangle_{n+1} \tag{2.12}$$
where c is a CHR constraint which has never been activate

Reactivate :
$$\langle [c\#i|A], S, B, T\rangle_n \longmapsto \langle [c\#i : 1|A], S, B, T\rangle_n \tag{2.13}$$
where c is a CHR constraint which is been added to A by *Solve*

Drop :
$$\langle [c\#i : j|A], S, B, T\rangle_n \longmapsto \langle A, S, B, T\rangle_n \tag{2.14}$$
where $c\#i : j$ has no occurrence j in the program

Simplify :
$$\langle [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \longmapsto \tag{2.15}$$
$$\langle \theta(C) +\!\!+ A, H_1 \uplus S, \theta \wedge B, T'\rangle_n$$
where the j^{th} occurrence of the CHR predicate of c in a (renamed apart) rule in the program is $r@H_1'\backslash H_2', d, H_3' \Leftrightarrow g|C$
and there exists a matching substitution θ such that
$c = \theta(d), chr(H_1) = \theta(H_1'), chr(H_2) = \theta(H_2'), chr(H_3) = \theta(H_3')$,
and $\mathcal{D}_b \models B \longmapsto \bar{\exists}_B(\theta \wedge g)$.
Let $T' = T \cup \{id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r]\}$
and $T' \neq T$

Propagate :
$$\langle [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \longmapsto \tag{2.16}$$
$$\langle \theta(C) +\!\!+ [c\#i : j|A, \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, \theta \wedge B, T'\rangle_n$$
where the j^{th} occurrence of the CHR predicate of c in a (renamed apart) rule in the program is $r@H_1', d, H_2' \backslash H_3' \Leftrightarrow g|C$
and there exists a matching substitution θ such that
$c = \theta(d), chr(H_1) = \theta(H_1'), chr(H_2) = \theta(H_2'), chr(H_3) = \theta(H_3')$,
and $\mathcal{D}_b \models B \longmapsto \bar{\exists}_B(\theta \wedge g)$.
Le $T' = T \cup \{id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3) +\!\!+ [r]\}$
and$T' \neq T$

Default :
$$\langle [c\#i : j|A], S, B, T\rangle_n \longmapsto \langle [c\#i : j+1|A, S, B, T\rangle_n \tag{2.17}$$
if in the current state no other transitions can fire

reinserts in the stack the constraints of the store that can be affected. Transition *Activate* corresponds to *Introduce* because it makes active (it assigns an occurrence to) the CHR constraint c at the top of the stack and inserts it in the store. *Simplify* and *Propagate* instead correspond to *Apply* since they fire a rule in presence of the matching constraints. If the active constraint does not match any rule then the active constraint becomes the next constraint in the stack (the occurrence number is incremented thanks to the *Default* transition). When the occurrence number of an active constraint is not present in the program *Drop* applies and the constraint is removed from the stack. Finally*Reactivate* is needed for activate again all the constraints that are moved from the store to the stack by a *Solve* transition.

Example 2.2.5. We report in Table 2.5 the same derivation presented in Example 2.2.4 but under ω_r instead of ω_t. For brevity the *built-in store* B and the *propagation history* T are omitted. As for the standard semantics the computation successfully terminates when no more transition rule can be applied. The only chr constraint remained in the store is the resulting constraint $gcd(3)$.

Even if the refined operational semantics reduces considerably the non-determinism of the standard one, it still contains sources of indeterminacy. The first is represented by the introduction of S_1 in the stack during a *Solve* transition: the introduction order of the constraints is indeed not specified. Furthermore when a *Simplify* or *Propagate* rule is applied the choice of the partner constraint (H_1, H_2, H_3) is not determined if more possibilities exist. The issue of removing the non-determinacy to realize practical CHR implementations is a source for optimizations of the constraint store representation and of the CHR execution.

Table 2.5: Example of derivation in refined semantics

$Initial\ state$	$\langle [gcd(6), gcd(9)], \emptyset \rangle_0$
$Activate$	$\langle [gcd(6)\#0 : 0, gcd(9)], \{gcd(6)\#0\} \rangle_1$
$Default$	$\langle [gcd(6)\#0 : 1, gcd(9)], \{gcd(6)\#0\} \rangle_1$
$Default$	$\langle [gcd(6)\#0 : 2, gcd(9)], \{gcd(6)\#0\} \rangle_1$
$Default$	$\langle [gcd(6)\#0 : 3, gcd(9)], \{gcd(6)\#0\} \rangle_1$
$Drop$	$\langle [gcd(9)], \{gcd(6)\#0\} \rangle_1$
$Activate$	$\langle [gcd(9)\#1 : 0], \{gcd(9)\#1, gcd(6)\#0\} \rangle_2$
$Default$	$\langle [gcd(9)\#1 : 1], \{gcd(9)\#1, gcd(6)\#0\} \rangle_2$
$Simplify$	$\langle [gcd(3)], \{gcd(6)\#0\} \rangle_2$
$Activate$	$\langle [gcd(3)\#2 : 0], \{gcd(3)\#2, gcd(6)\#0\} \rangle_3$
$Default$	$\langle [gcd(3)\#2 : 1], \{gcd(3)\#2, gcd(6)\#0\} \rangle_3$
$Default$	$\langle [gcd(3)\#2 : 2], \{gcd(3)\#2, gcd(6)\#0\} \rangle_3$
$Propagate$	$\langle [gcd(3), gcd(3)\#2 : 2], \{gcd(3)\#2\} \rangle_3$
$Activate$	$\langle [gcd(3)\#3 : 0, gcd(3)\#2 : 2], \{gcd(3)\#3, gcd(3)\#2\} \rangle_4$
$Default$	$\langle [gcd(3)\#3 : 1, gcd(3)\#2 : 2], \{gcd(3)\#3, gcd(3)\#2\} \rangle_4$
$Simplify$	$\langle [gcd(0), gcd(3)\#2 : 2], \{gcd(3)\#2\} \rangle_4$
$Activate$	$\langle [gcd(0)\#4 : 0, gcd(3)\#2 : 2], \{gcd(0)\#4, gcd(3)\#2\} \rangle_5$
$Simplify$	$\langle [gcd(3)\#2 : 2], \{gcd(3)\#2\} \rangle_5$
$Default$	$\langle [gcd(3)\#2 : 3], \{gcd(3)\#2\} \rangle_5$
$Drop$	$\langle [], \{gcd(3)\#2\} \rangle_5$

2.2.2 Important CHR properties

In this section we will discuss the properties of *confluence*, *anytime*, *online* and *concurrency*. As it will be shown confluence does not guaranty only the uniqueness of the final state but it is a very important property that can help the parallelization of CHR program. The latter three properties, instead, are automatically embedded in any algorithm described by a CHR specification. They will be analyzed from the point of view of the very abstract semantics and the limitations for the standard and refined ones will be pointed out. The coverage of this subject follows what reported in Chapter 4 and Chapter 5 of [46].

Confluence

Confluence means that if there is more than one way to rewrite constraints they all must yield the same result. In other words confluence guarantees that, given a goal, any possible computation must result in the same final state no matter the order of the applicable rules. To formally define confluence we introduce the concept of *joinability*. Denoting the reachability relation $\longmapsto^*$ as the reflexivity transitive closure of $\longmapsto$, two states S_1 and S_2 are said to be joinable if there exist two states S'_1 and S'_2 such that $S_1 \longmapsto^* S'_1$, $S_2 \longmapsto^* S'_2$ and $S'_1 \equiv S'_2$. Basically this definition states that two states are joinable if there are two computation that applied on these states can lead to the same result. Now we are ready to formally define confluence [4]:

> a CHR program is said to be *confluent* if, for all the states S, S_1 and S_2 of its computation, the following implication holds:
> if $\quad S_1 \longmapsto^* S_1$ and $S_2 \longmapsto^* S_2 \quad$ then $\quad S_1$ and S_2 are joinable

As we have seen discussing the refined semantics, confluence is a very important property for a CHR program, hence it is important to point out that a decidable, sufficient and necessary confluence test for terminating programs exists. Indeed, if the program is terminating, the search for joinability can be restricted to a finite number of relevant states called *overlaps*. Overlaps are state where several rules are applicable possibly leading to different states. Given a couple of rules for each overlap we have two possible resulting states named *critical pair*. If a critical pair cannot be joinable the program is not confluent. The sufficient and necessary condition for a terminating program to be confluent is given by the condition that all its critical pairs must be joinable [1].

Example 2.2.6. The running Exemple 2.2.1 is confluent when the arguments are fully known (ground confluent) but it is not confluent in general. Indeed, for instance, the simple overlap:

$$gcd(A) \land gcd(B) \land gcd(C) \land A \leq B \land A \leq C$$

generates the critical pair:

$$\begin{array}{ll} gcd(A) \wedge gcd(B - A) \wedge gcd(C) \wedge A \leq B \wedge A \leq C & \text{and} \\ gcd(A) \wedge gcd(B) \wedge gcd(C - A) \wedge A \leq B \wedge A \leq C & \end{array}$$

that cannot be joinable until the two numbers remain unknown or a further relationship between variables is not added.

Finally another good feature of confluence should be mentioned: in some case if a CHR program is not confluent it can be incremented by the addition of new rules in order to make it confluent. Such technique is called completion [3]. The new rules are generated by each critical pairs (usually one propagation and one simplification are needed) and in turn they may generate other non joinable critical pairs. Hence the completion process may not be terminating.

Anytime

An algorithm is called an *anytime* algorithm when, interrupted at any time, it can be resumed starting from the intermediate result without the need to restart from scratch. If this property holds the result of the partial execution becomes an approximation of the final result and further interrupts of the algorithm will lead to a better approximation (closer to the final result). CHR embeds the *anytime* property in its semantics since the interruption of the execution after any rule application gives the current *state* of the computation. Any state of the transition system can be resumed without the need of restarting from the initial state and any further state approaches the final state since more built-in constraints are added and the CHR constraints are even further rewritten.

Example 2.2.7. In Example 2.2.3 we have seen an execution of the gcd program of Example 2.2.1. We can easily notice that, if we halt the computation in a given state and then we use the resulted constraints as a query for a new execution of the same program, the same final result will be obtained.

In the standard and refined semantics things are more tricky because, if we retrieve the logical reading of a state after an interrupt of the execution, we loose the propagation history and the distinction between goal constraints and their numbered or active instance. However for the standard semantics, it is still possible to restart the computation from the logical reading of a state after some executions of *introduce* and *solve* that can lead to the state itself. While for the highly deterministic refined semantics the property of *confluence* is required in order to restart the computation from the logical reading of a state.

Online

In order to understand the online property we need to introduce the concept of monotonicity. The monotonicity property of CHR states that if the application of a transition rule is allowed for a given state then it is allowed in a state that contains additional constraints as well. The following implication resumes the property of monotonicity over the conjunction of constraints A, B and E.

$$\text{if } A \longmapsto B \text{ then } A \wedge E \longmapsto B \wedge E \tag{2.18}$$

The validity of such property for ω_a is evident: since the states are just goals, the addition of a new conjunction of constraints can be seen as an increment of the term G in the *apply* rule (Table 2.1). If the condition $\mathcal{D}_b \models G \longmapsto \bar{\exists} C$ holds, $\mathcal{D}_b \models G \wedge E \longmapsto \bar{\exists} C$ will hold as well. In ω_t monotonicity is still present if we add to a non-failed state the suitable constraints to all its first three components and we remove entries from the propagation history. In ω_r the constraints can be added only "at the end" of the stack, and the restriction for numbered and active constraints must be respected. Thus it is necessary to pay attention to the fact that there are two possible ways for combining two states.

A direct consequence of monotonicity is the online property. An algorithm is said to be online if it allows for the addition of new constraints during its execution without the need of restart computation from the beginning. Indeed monotonicity implies that constraints can be incrementally added to the states of the execution of a program resulting in final state equivalent to the one originated by the execution that has those constraints from the initial state.

Example 2.2.8. The online property can be observed comparing the two execution in Table 2.6 of the gcd program 2.2.1, in one case with the initial query $\{gcd(6), gcd(9), gcd(2)\}$ and in the other one with $\{gcd(9), gcd(6)\}$ plus the addition of constraint $gcd(2)$ (highlighted in red) at an intermediate state.

Concurrency

Concurrency is defined as the ability of allowing logically independent computations that are capable of composing the final system in a modular fashion. This does not necessary imply that the concurrent processes act simultaneously (in parallel). Hence concurrency can benefit by parallelism for improving performance, but can also implemented in a sequential way. If a suitable parallel hardware is available concurrency can result in parallel semantics, otherwise, in presence of a single computation engine, it leads to an interleaving semantics. Declarative programming languages in general are more oriented towards concurrency than imperative because different computations can be composed together without conflicts. In particular in CHR concurrent processes are seen as atomic CHR constraints that communicate asynchronously through a shared built-in constraint store. The built-in constraints are the messages and the variables are the communication channels.

The monotonicity property 2.18, introduced in the previous paragraph, is at the basis of the parallelism of CHR [45]. Indeed from monotonicity it is possible to derive the notion on *weak parallelism*: separate parts of the problem can be tackled

Table 2.6: Example of online property

$\langle gcd(6) \wedge gcd(9) \wedge gcd(2)\rangle$	$\langle gcd(6) \wedge gcd(9)\rangle$
$\langle gcd(6) \wedge gcd(3) \wedge gcd(2)\rangle$	$\langle gcd(6) \wedge gcd(3)\rangle$
$\langle gcd(4) \wedge gcd(3) \wedge gcd(2)\rangle$	$\langle gcd(3) \wedge gcd(3)\rangle$
$\langle gcd(1) \wedge gcd(3) \wedge gcd(2)\rangle$	$\langle gcd(0) \wedge gcd(3)\rangle$
$\langle gcd(1) \wedge gcd(1) \wedge gcd(2)\rangle$	$\langle gcd(0) \wedge gcd(3) \wedge gcd(2)\rangle$
$\langle gcd(0) \wedge gcd(1) \wedge gcd(2)\rangle$	$\langle gcd(3) \wedge gcd(2)\rangle$
$\langle gcd(0) \wedge gcd(1) \wedge gcd(1)\rangle$	$\langle gcd(1) \wedge gcd(2)\rangle$
$\langle gcd(1) \wedge gcd(1)\rangle$	$\langle gcd(1) \wedge gcd(1)\rangle$
$\langle gcd(0) \wedge gcd(1)\rangle$	$\langle gcd(0) \wedge gcd(1)\rangle$
$\langle gcd(1)\rangle$	$\langle gcd(1)\rangle$

in parallel. Formally weak parallelism is defined in Figure 2.2(a) where A, B, C and D are conjunctions of constraints, $\Rrightarrow$ denote a parallel transition and the upper transitions imply the lower one.

Figure 2.2: Weak parallelism (a) and trivial confluence (b)

$$\frac{\begin{array}{c} A \longmapsto B \\ C \longmapsto D \end{array}}{A \wedge C \Rrightarrow B \wedge D} \qquad\qquad \frac{\begin{array}{c} A \longmapsto B \\ C \longmapsto D \end{array}}{A \wedge C \Rrightarrow A \wedge D \text{ or } B \wedge C}$$

(a) (b)

To prove weak parallelism a direct consequence of monotonicity, called *trivial confluence*, is needed. In Figure 2.2(b) we can see that an intermediate state of the parallel computation of $A \wedge C$ can be either $A \wedge D$ or $B \wedge C$. Thus from each of these states $B \wedge D$ can be easily obtained. Weak parallelism is then proved in an interleaving semantics composing two computations.

Example 2.2.9. Weak parallelism can be applied to the running Example 2.2.1 on different part of the query. In Table 2.7 a derivation from the query $gcd(2)$, $gcd(4)$, $gcd(9)$, $gcd(6)$ is reported. For instance in the first transition rule `R1` is applied concurrently to $gcd(2) \wedge gcd(4)$ and to $gcd(9) \wedge gcd(6)$

Since weak parallelism is based on monotonicity when applied in ω_t or ω_r the

Table 2.7: Example of computation that employs weak parallel transitions

$\langle gcd(2) \wedge gcd(4)$	$\wedge$	$gcd(9) \wedge gcd(6)\rangle$	$\mathrel{\vDash\!\!\Rrightarrow}$
$\langle gcd(2) \wedge gcd(2)$	$\wedge$	$gcd(3) \wedge gcd(6)\rangle$	$\mathrel{\vDash\!\!\Rrightarrow}$
$\langle gcd(0) \wedge gcd(2)$	$\wedge$	$gcd(3) \wedge gcd(3)\rangle$	$\mathrel{\vDash\!\!\Rrightarrow}$
$\langle gcd(2)$	$\wedge$	$gcd(0) \wedge gcd(3)\rangle$	$\longmapsto$
$\langle gcd(2)$	$\wedge$	$gcd(3)\rangle$	$\longmapsto$
$\langle gcd(2)$	$\wedge$	$gcd(1)\rangle$	$\longmapsto$
$\langle gcd(1)$	$\wedge$	$gcd(1)\rangle$	$\longmapsto$
$\langle gcd(0)$	$\wedge$	$gcd(1)\rangle$	$\longmapsto$
$\langle gcd(1)\rangle$			

same limitation imposed by such property should be applied. It is worth noting that, in any case, weak parallelism cannot be applied on the same constraint: one constraint can be used at most once in a simultaneous computation. Moreover weak parallelism cannot be applied to rules that share built-in constraints for the guard check. In order to overcome these limitation the notion of *strong parallelism* is introduced. As it can be noticed from Figure 2.3(a) strong parallelism allows rules to work also on overlapping constraints provided that they do not attempt to modify them. If the overlapping parts E is kept by the two rules they can work

Figure 2.3: Strong parallelism (a) and trivial confluence with context (b)

$$\frac{\begin{array}{c} A \wedge E \longmapsto B \wedge E \\ C \wedge E \longmapsto D \wedge E \end{array}}{A \wedge E \wedge C \mathrel{\vDash\!\!\Rrightarrow} B \wedge E \wedge D} \qquad \frac{\begin{array}{c} A \wedge E \longmapsto B \wedge E \\ C \wedge E \longmapsto D \wedge E \end{array}}{A \wedge C \wedge E \mathrel{\vDash\!\!\Rrightarrow} A \wedge D \wedge E \text{ or } B \wedge C \wedge E}$$

(a) (b)

concurrently. As in the case of weak parallelism the proof of strong parallelism descend from monotonicity and trivial confluence, but, in this case, with context (Figure 2.3(b)). The domain of application of strong parallelism consists of all the propagation rule since they just add new constraints and some simpagation rules because part of their head is kept. It is clear that simplification rules cannot take advantage of strong parallelism since they remove all the constraints in their head.

Example 2.2.10. Continuing with the running Example 2.2.1, in Table 2.8 a strong parallel derivation of the query $gcd(2), gcd(4), gcd(9), gcd(6)$ is shown. In the first state the constraint $gcd(2)$ matches the kept part of rule `R1` while all the other constraints match in turn the removed part. Thus the first strong parallel transition corresponds to the concurrent application of three sequential transitions.

Table 2.8: Example of computation that employs strong parallel transitions

$\langle gcd(2)$	$\wedge$	$gcd(4) \wedge gcd(9) \wedge gcd(6)\rangle$	$\mapstochar\rightrightarrows$
$\langle gcd(2)$	$\wedge$	$gcd(2) \wedge gcd(7) \wedge gcd(4)\rangle$	$\mapstochar\rightrightarrows$
$\langle gcd(2)$	$\wedge$	$gcd(0) \wedge gcd(5) \wedge gcd(2)\rangle$	$\mapstochar\rightrightarrows$
$\langle gcd(2)$	$\wedge$	$gcd(3) \wedge gcd(0)\rangle$	$\mapstochar\rightrightarrows$
$\langle gcd(2)$	$\wedge$	$gcd(1)\rangle$	$\longmapsto$
$\langle gcd(1)$	$\wedge$	$gcd(1)\rangle$	$\longmapsto$
$\langle gcd(0)$	$\wedge$	$gcd(1)\rangle$	$\longmapsto$
$\langle gcd(1)\rangle$			

Parallelization under ω_a or ω_t is possible with the limitation that we have just seen, but the adoption of ω_r will lead to incorrect behaviors since it strongly relies on the order of rules in the program and on the order of the goal constraints. However, as in the case of anytime property, *confluence* helps the parallelization in ω_r as well. Indeed such property guarantees that the order of the rules and thus even the order of the constraints in the goal does not matter. Hence, if the program is confluent, it can be executed in parallel even if it is written for the refined semantics.

2.2.3 Parallel execution of CHR

As we have shown in the previous Section CHR is a highly concurrent language. Likewise it is broadly accepted that a parallel computation model is still *in fieri*. The first example of parallel implementation was due to Frühwirth [45] where it was shown how to evaluate the degree of concurrency starting from the confluence analysis of a sequential program execution. Further works by Sulzmann and Lam [100, 68] focus on the formal specification and the development of a parallel implementation of the CHR goal-based execution schema: multiple processor cores run multiple threads following a single CHR goal. The novel semantics introduced in these latter papers can be considered the direct extension of the refined operational semantics towards parallelism and hence the most suitable for a concrete and efficient parallel implementation.

Other attempts to concurrency were pursued in the last years mainly driven by the CHR set based operational semantics [88]. Although CHR programs usually adopt a multiset based semantics it was shown how a large class of programs can benefit from a tabled rule execution schema that eliminates the need of a propagation history and acquires a natural parallelism by the notion of set. The persistent constraint semantics presented in [15], that exploits the idea of a mixed store where the constraints can behave like a set or a multiset, achieves a higher degree of declarativity keeping the potentiality of concurrency of the standard semantics. Finally we should mention the parallel execution strategy, introduced in [84], that gives the possibility of applying multiple removals to the same constraint. Such semantics eliminates the conflicts in the constraint removals by different instances of the same rule remaining sound wrt the sequential execution.

3 Reconfigurable Computing

Reconfigurable computing is a particular computer architecture able to merge the flexibility of software with the performance of hardware using as processing element a high speed configurable hardware fabric [56]. If compared with traditional microprocessor architecture, the main difference is represented by the ability of modifying not only the control flow but also the data path.

In this chapter we want to introduce the hardware component employed in all the experimental setups described along this thesis: the Field-Programmable Gate Array (FPGA) [25, 67]. FPGAs are a clear instance of reconfigurable computing: the fist section of the chapter explains what makes them programmable. Then in Section 3.2 we discuss the methodology for deploying hardware designs to FPGA, the process called synthesis. In the last Section 3.3 our focus will be the state of the art of the high-level languages employed in the hardware synthesis of reconfigurable systems.

3.1 Field-Programmable Gate Arrays

FPGAs are considered by hardware engineers as one of the most useful component/tool in digital electronics. The key factor that brought FPGAs to success was its programmability. Indeed FPGAs are hardware devices that consist of arrays of logic gates capable of being programmed by the customer in the field rather than in a semiconductor fabrication plant. The drawbacks given by this flexibility are mainly two: the grater amount of space occupied by a project in FPGAs and the maximum speed achievable. Since our work is more concerned with the research of suitable hardware configurations than an effective gain in hardware performance, the FPGA is the most suitable hardware device to be used as circuits test bench. In the next Sections we inspect the main components of an FPGA while in Section 3.1.5 the main differences between an FPGA and a standard integrated hardware circuit are pointed out.

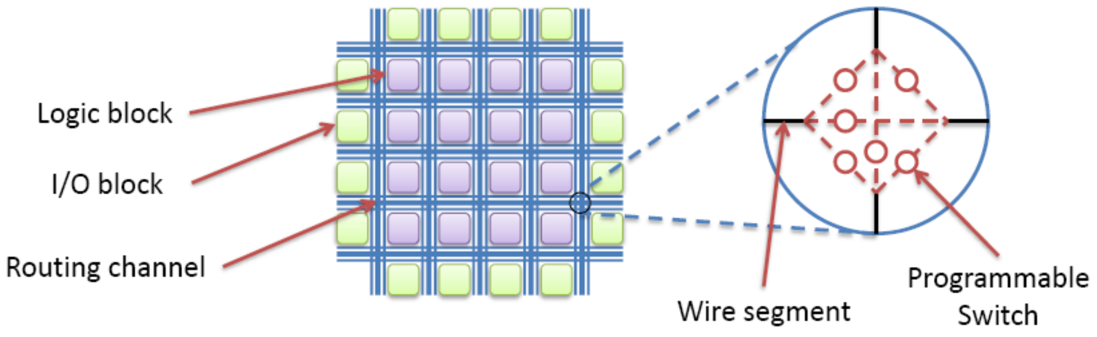


Figure 3.1: FPGA architecture (island style)

3.1.1 FPGAs Architecture

FPGAs are hardware devices consisting of arrays of logic gates capable of being programmed by the customer in the field rather than by the manufacturer. They are devices containing programmable interconnections between logic components, called *logic blocks*, that can be programmed to perform complex combinational functions. In most FPGAs the logic blocks contain memory elements like simple flip-flops or complete blocks of memory. FPGAs can also host hardware components like embedded hard microprocessors or IP (Intellectual Property) cores that use the logic blocks themselves to make predefined structures like soft microprocessor cores, real CPUs entirely implemented using logic synthesis.

The architecture of an FPGA is basically divided into three parts: the internal computational units called configuration logic blocks (CLBs), the Input/Output (I/O) blocks that are responsible for the communication with all the other hardware resources outside the chip, and the programmable interconnections among the blocks. Besides these essential blocks in the modern FPGA we can find also more rich and complex hardware blocks designed to perform higher-level functions (such as adders and multipliers), or embedded memories, as well as logic blocks that implement decoders or mathematical functions.

The FPGA architecture is model and vendor dependent, but in the most cases it consists of a bi-dimensional array of CLB surrounded by the I/O blocks. In the sketch of Fig. 3.1 it is shown this typical arrangement of the internal blocks called island-style. It follows a detailed description of the CLB, I/O block and interconnections.

3.1.2 Configuration Logic Block

CLBs are the flexible logic part of an FPGA, they can implement any logic function, if the number of them is sufficient. CLBs are made by few logic cells commonly called *slices* and a full crossbar for interconnection. Each of them consists of some n-bit lookup tables (LUT), full adders (FAs) and D-type flip-flops (see Fig. 3.2). The n-bit LUT performs the combinatorial part of the circuit: it can encode any n-input boolean function by a truth table model. The FA is used when there is the need to perform arithmetic functions otherwise it can be bypassed by a multiplexer. Likewise the final D flip-flop can be skipped if we wish an asynchronous output. Instead the LUT can be bypassed for executing just the sequential part (the flip-flop) by programming its output to simply reproduce its input. The crossbar is responsible for the connection among the inputs of the CLB and any input of the slices and for guaranteeing the feedback mechanism of the output of the slices. The output of the slices instead is connected directly to one of the outputs of the CLB.

Beside this basic configuration, more complicate structures recently appear. For example, many FPGAs can contain multiple LUTs. Increasing the number of inputs, n, the quantity of logic implementable in a CLB increases giving as result the possibility of decrease the number of connection among them. However since the complexity of a LUT increases exponentially with n as well, it is better to merge multiple small LUT with a local routing [8][110]. In Fig. 3.2 it is shown how to implement a 5-input LUT starting from two 4-input ones. Given a function that requires 5 inputs, the first LUT implements the function assuming '0' as logic value for the last input while the other LUT does the same assuming '1'. The fifth input is routed to the select input of a multiplexer that can choose the output of the two LUTs.

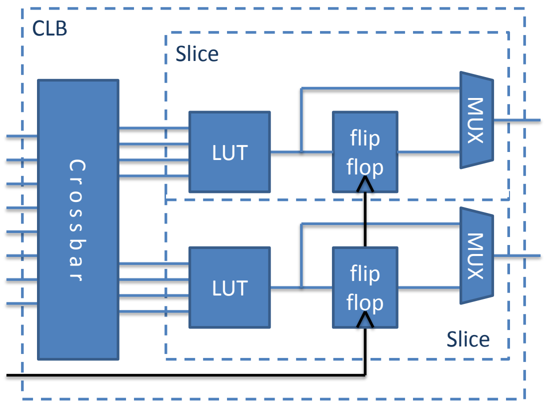
3.1.3 Routing system and I/O blocks

In the island-style architecture the CLBs are separated by rows and columns of routing channels [111]. Each of them spans the entire length of the array and their portions around the blocks are called channel segments. The segment channel is composed by many segment wires grouped together and connected by programmable switch (see the expanded view of Fig. 3.1) that allows wires to turn corner or to extend further down a channel. Usually such routing switches are implemented by using multiplexer that chooses the driver of a segment wire.

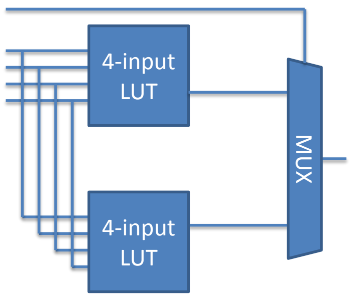
I/O blocks can be programmed to be inputs, outputs or three-states and are generally programmed to implement other features like low power consumption or high speed connections. The blocks contain a pin that physically bridges the FPGA with the outer routing. In order to match the I/O characteristics of the other devices witch are interfaced with the FPGA, the I/O block can support a wide variety of I/O programmable standards as well.

3.1.4 Embedded components

In an FPGA architecture the manufacture can choose to embed specialized hardware blocks. These fixed structures typically implement common function used in the majority of the digital design with an optimized layout that reduces the occupied



(a)



(b)

Figure 3.2: (a) CLB architecture (b) Two 4-input LUT become a 5-input LUT

area and also the execution time with respect to the same function implemented in CLBs [66]. Since the number and the size (in terms of inputs) of such specialized blocks are not predictable in advance by the manufacture, the strategy usually adopted is to split them in smaller sub-block. The hardware components that better apply to fragmentation and hence can be frequently found in FPGAs are multipliers and memory blocks. In particular in Chapter 6 we will show how we have employed memory blocks for guaranteeing a safe decoupling interface between the central CPU and the custom hardware running on FPGA.

A particular class of specialized hardware blocks deserves a special attention: the embedded processor [41]. Indeed it is possible to embed both a soft or a hard microprocessor in FPGAs. The former is implemented using CLBs while the latter is built from dedicated silicon. Typical instances of hard physical processors that can be found in FPGAs come from the PowerPC or ARM families. The most evident advantage in adopting an embedded processor is the high degree of freedom in customization. The designer can invent new and unique peripherals that are easily positionable on the processor bus. Another benefit introduced by the coupling of FPGAs and processors is the ability of making a trade off between hardware and software and, as a consequence, the possibility of demanding some software bottleneck to specialized hardware engine. In Chapter 6 the design of such coupled system is fully discussed and a practical implementation is given. Here we focus on pointing out the lack of suitable tools for embedded processor software design due to the fact that studies on such field are relatively immature compared with the ones of software or FPGAs design. Finally the sole advantage of the soft embedded microprocessor is its non-obsolescence since its design is easily portable on other reconfigurable hardware.

3.1.5 FPGA vs. ASIC

What makes the real success of FPGAs over their counterparts "the Application Specific Integrated Circuits" (ASICs) is clearly identifiable in their reconfigurability [66]. Since FPGAs guarantee a time to production extremely shorter, they can be programmed and verified in few weeks. We can, thus, easily understand why they quickly emerged as a way for generating effective and low cost hardware prototype. However nowadays FPGAs are not used only for prototyping, indeed, due to the decreasing cost per gate, they are employed as a principal component in many digital hardware designs. In the following list we summarize the differences between an FPGA and a classical ASIC.

- Usually in FPGAs the primitive components available are more complete and complex of the one provided by the ASIC manufacturing. Depending on the situation this may or may not represent an advantage because in FPGAs a not used resource is wasted space. The ASIC designer instead can choose to implement or not the primitives at his disposal.

- FPGAs are able to work only at relatively slow speed compared to ASICs. The maximum clock frequency is strongly affected by the non optimized structure of the project layout in the FPGA. If the degree of parallelization achievable in FPGAs is not enough for the time requirement of the hardware design the only solution is to move the project on ASIC.

- FPGAs have a low density in terms of logical circuitry. The flexibility in the interconnection and the general structure of a CLB makes the effective capacity of an ASIC equal to the one of several FPGAs. This has also the drawback of requiring more space in the silicon wafer for producing FPGAs.

- As we said prototyping costs and production time are more convenient for FPGAs. Indeed FPGAs are usually employed in the *emulation* stage of the ASICs production. Functional bugs can be quickly pointed out moving an ASIC circuit on an FPGA and using it as an emulator.

3.2 Programming reconfigurable systems

FPGAs and reconfigurable architecture in general have the user programmability typical of software, but also the spatial parallelism of hardware. To get full advantage from them, a programming model including a complete framework, that allows configuration changes, is needed. Such infrastructure has not only to guarantee the temporal programmability but also to deal with spatial issues, like physical placement or timing of functional units, that are usually considered hardware properties.

The next section provides an introduction to the design flow of a reconfigurable hardware. We will cover the FPGAs synthesis from the programming phase through the compilation stages to the actual code deployment in hardware. In Section 3.2.3 a brief overview of a typical hardware programming language (namely VHDL) is given. VHDL is for hardware what assembly is for software with the difference that it can be called "semi-portable" since the code is not completely FPGA dependent. It let the programmer have a fine control of hardware and parallelism but it requires to pay attention to many low-level details.

3.2.1 FPGA configuration

The FPGA programmer usually begins the design flow by a description of the desired behavior of the targeting hardware in a hardware description language (HDL). The most commonly adopted HDLs by the hardware engineers are Verilog [108] and VHDL [109] (in particular we will use the latter in all the implementations proposed in this thesis). HDL code can directly feed a vendor-specific design automation tool (called *synthesizer*) that through different steps generates a technology-mapped netlist used to configure each CLB. Since each FPGA differs from design architecture a dedicated process, named place-and-route, takes care of choosing which CLBs need to be employed and how to physically connect them. Before the actual implementation in hardware, the programmer can validate the map via timing analysis, simulation, and other verification methodologies. The final result of the design flow is then a bit-stream file that can be transfered via a serial interface to the FPGA or to an external memory device charged to deploy it at every boot of the FPGA.

In the following Sections a deeper description of each synthesis step is given. Here we report just a very simple example of the RTL and netlist level that could be useful for having an idea of the different abstraction levels.

Example 3.2.1. The two schematics reported in Fig. 3.3 are the circuital drawings of a 2-bit counter at the RTL level (a) and at the netlist level (b). We consider as 2-bit counter a very simple hardware component with no input and two outputs that perform in a loop the sequence 00, 01, 10, 11. The RTL structural description deals with the elementary hardware block of the combinatorial logic (in the example a *NOT* and a *XOR* gate) and of the sequential logic (two D-type flip-flops), while the netlist level makes explicit all the interconnections among the gates.

3.2.2 Register transfer level

Register transfer level (RTL) usually is the first level of abstraction encountered in the synthesis flow. RTL descriptions are then compiled to a gate-level by a logic synthesis tool. As the name suggests, at this stage the design is described in terms of transfer of data among hardware registers and through combinatorial logic elements. The most common way to implement registers are the D flip-flops, elementary memory blocks capable of storing the value of a signal. In a D flip-flop every clock cycle the value of the output is replaced by the value of the input.

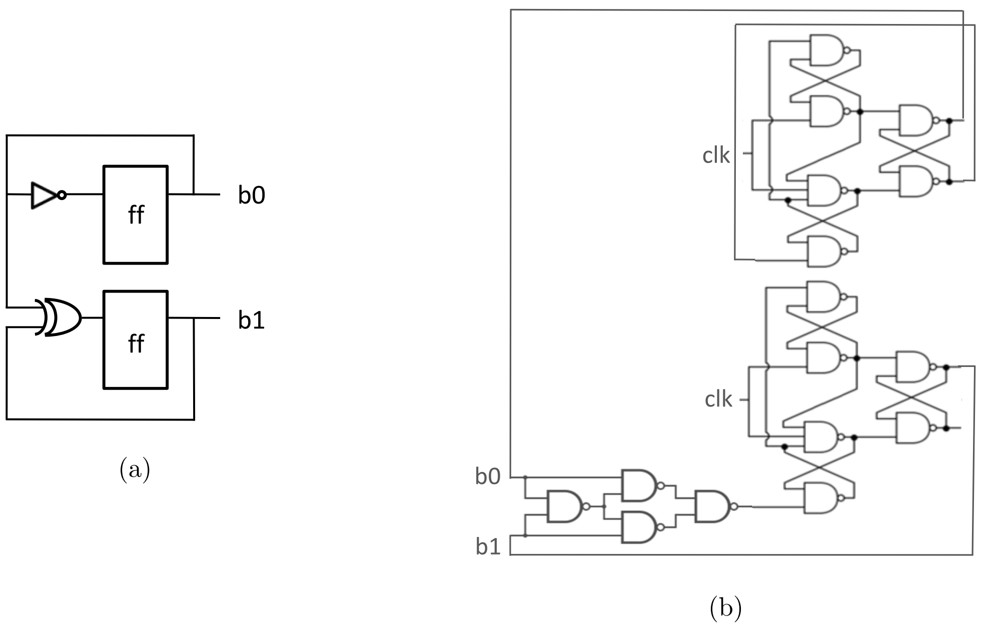


Figure 3.3: 2-bit counter seen at the RTL level (a) and at the netlist level (b)

Such register elements have a twofold duty: they synchronize the operations of the circuit to the clock edge and store the information of the previous clock cycle. The combinatorial part of the circuit is, instead, performed by logic gates (i.e. AND, OR, NOT, XOR, XNOR, etc.) that can implement any Boolean functions.

As we said the design of the RTLs is commonly done using a HDL: the designer declares the register and describes the interconnections among combinatorial logic using standard constructs like if-then-else and arithmetic operations.

Example 3.2.2. In Figure 3.4 the circuit described in Example 3.2.1 is specified using VHDL in a structural way (see Section 3.2.3 for more details) that highlights the registers and logic gates interconnections. For clarity reason the entity part of the code has been omitted, and only the architecture is reported.

Using an electronic design automation (EDA) tool for synthesis, such specification can be directly translated to an equivalent hardware implementation file for FPGAs or ASICs. The right part of the figure presents a behavioral architecture of an inverter and a xor gate, but their declaration is not necessary for the synthesis process if the components are stored in one library or package defined by the hardware vendor.

It is worth noting that at RTL some types of hardware circuits can be recognized. For example if there is a cyclic path of logic from a set of outputs of some registers to its inputs, the circuit is converted in a state machine. Otherwise if there are logic paths from a register to another without a cycle the path is converted in a pipeline.

Figure 3.4: 2-bit counter structural description

```
architecture STRUCTURE of COUNTER2 is
  component DFF
    port(CLK, RESET, DATA: in BIT;
    Q: out BIT);
  end component;
  component INV
    port(I: in BIT;
    O: out BIT);
  end component;
  component XOR2
    port(I1, I2: in BIT;
    O: out BIT);
  end component;

  signal N0, N1, N2, N3: BIT;
begin
  u1: DFF port map(CLK, N0, N1);
  u2: DFF port map(CLK, N2, N3);
  u3: INV port map(N1, N0);
  u4: XOR2 port map(N1,B3, N2);
  COUNT(0) <= N1;
  COUNT(1) <= N3;
end STRUCTURE;
```

```
architecture BEHAVIORAL of DFF is
begin
  process(CLK, RESET)
  begin
    if RESET='1' than
      Q <= '0';
    elsif CLK'event and CLK='1' then
      Q <= DATA;
    end if;
  end process;
 end BEHAVIORAL;

architecture BEHAVIORAL of INV is
begin
  O <= not I;
end BEHAVIORAL;

architecture BEHAVIORAL of XOR2 is
begin
  O <= I1 xor I2;
end BEHAVIORAL;
```

3.2.3 VHDL

VHDL stands for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language. It was originally developed in 80s by the US Department of Defense for describing ASICs, and nowadays, together with Verilog, is the most used programming language for digital circuits.

Programming in VHDL has some analogy with object-oriented language like encapsulation and interfaces but it differs from imperatives language like C++ or Java because of its concurrent semantics [56]. Indeed VHDL is inherently parallel: commands (that correspond to instantiation of logic gates) are computed in parallel as soon as new inputs arrive.

There are two levels of abstraction that can be used in HDL for describing hardware: the structural and the behavioral one. The first describes the system in terms of interconnections between hardware components composed by elementary hardware primitives, while the second works at a higher level of abstraction dealing with the functionality of the hardware component, or, in other words, it describes what the component actually do specifying the relation between the input and the output signals.

Example 3.2.3. In Figure 3.5 is reported the behavioral description of the 2-bit counter architecture presented in Example 3.2.2 in its structural version.

Figure 3.5: 2-bit counter behavioral *architecture*

```
architecture BEHAVIORAL of COUNTER2 is
  signal tmp : BIT_VECTOR(1 downto 0);
begin
  process(clk, reset)
    if reset='1' then
      tmp <= "00"
    elsif clk'event and clk='1' then
      tmp <= tmp+1;
    end if;
  end process;
  COUNT <= Tmp;
end BEHAVIORAL;
```

We have seen how signals can be used as wire for connecting component ports, but in this example they are used for representing logic. The operator `<=` is used for assigning a value to a signal or for defining the signal as a function of other signals. The VHDL operator for assignment can seem similar to the C `=` but the main difference is that the assignment is not "instantaneous" and it takes effect only at the next clock cycle. The VHDL operator most similar to the C equality is `:=` but it cannot be used for signals (only for constants and variables). Instead nothing changes for the relational equality operator represented in C with `==` and in VHDL with `=`.

After an instantiation of the signals used in the *architecture* concurrent signal assignments can be used inside the *begin/end* statement but outside the *process* block. Instead the content of a *process* is executed in sequence and its execution is triggered by the value changing of a signal inside the *sensitivity list* (the list between brackets that follows the process statement). The *process* is suspended when all the statements inside it are executed and signals are assigned only at this time marking the begin of a new clock cycle. Indeed the signals retain the logical values of the former clock cycle until the end of the *process* execution.

Besides the data types that we briefly introduce in the next example, a certain number of *attributes* are defined. They are used in conjunction with signals or variables and they return various types of information about them. In the present example the *event* attribute is used for the `clk` signal. In Table 3.1 a complete list of them is given.

It is worth pointing out that the signal assignments and the execution of a *process* are concurrent statements. This implies that the statements are executed when one or more of the signals on the right hand side or in the *sensitivity list* change their value. There may be a propagation delay associated with this change. Digital systems are basically data-driven and an event which occurs on one signal will lead to an event on another signal, etc. The execution of the statements is determined

Table 3.1: Defined *attributes* for signals and variables

Attribute	*Function*
event	returns the Boolean "true" if any kind of event on the signal occurred (like a value changing)
active	returns the Boolean "true" if there has been an assignment on that signal
transaction	returns a"bit" signal that toggles every time there is a transaction on that signal
last_event	returns the time interval since the last event on that signal
last_ active	returns the time interval since the last transaction on that signal
last_value	gives the previous value of a signal
delayed(T)	gives a T delayed version of a given signal
stable(T)	returns the Boolean "true" if no actions on that signal for an interval T
quiet(T)	returns the Boolean "true" if no transactions on that signal for an interval T

by the flow of signal values. As a result, the order in which these statements are given does not matter. This is in contrast to conventional, software programs that execute the statements in a sequential or procedural manner.

Besides the *architecture* that we have seen in the previous examples another structure is mandatory for a complete VHDL code: the *entity*. The description of an hardware module requires an *entity* declaration that indicates all the interfacing signals to the outer word. It is a list of the I/O ports of the hardware block: it specifies the direction, the type, the bit width and the endianness of the signals.

Example 3.2.4. The following Figure 3.6 reports the *entity* declaration of the running Example 3.2.2.

Figure 3.6: 2-bit counter *entity*

```
entity counter is
  port( clk, reset : in  BIT;
        COUNTER    : out BIT_VECTOR(1 downto 0));
end counter;
```

Since VHDL is a strongly typed language the programmer has always to declare the type of every signals, constants or variables. The type can be built-in or user defined. Examples of them are: BIT that can assume the values '0' or '1';

standard_logic that can assume 9 values depending by the strength of the signal (reported in Table 3.2.3; or boolean that can have the value "true" or "false". Each type can be also assigned to a group of signals using the extension _vector.

In his example the *entity* is called *counter* and has two inputs of type BIT and one output of type BIT_VECTOR of width 2 and with the most significant bit as the first bit.

Table 3.2: std_logic type as defined in the std_logic_1164 package

type STD_LOGIC is (	
'U'	uninitialized
'X'	forcing unknown
'0'	forcing '0' value
'1'	forcing '1' value
'Z'	high impedance
'W'	weak unknown
'L'	weak '0'
'H'	weak '1'
'-');	don't care

3.2.4 FPGA physical synthesis

The synthesis of an FPGA is a multi-step process that begins with a logical synthesis, translates the HDL code at the RTL to a netlist at the gate level, applies a place and route algorithm, maps the generic netlist to a target device, and finally generates and deploys the bitstream to the FPGA. The whole sequence is called physical synthesis. It is a process entirely performed by a tool, provided by the FPGA manufacturer, that brings the HDL to a placed and routed netlist without user intervention. Besides the synthesizable HDL code, the fundamental inputs that the user has to provide for the synthesizer can be summarized as follows:

- Model of the target family and device;
- The clock frequency (or frequencies for multiple clocks design) with respect to an external clock;
- The required output timing;
- The output driver standard and signal strength;
- The output slew rate;
- The map between physical output pins and output signals;

- The maximum acceptable time delay between groups of elements.

In the following paragraphs each step taken by the synthesizer is briefly described.

Logical synthesis

With the expression "logical synthesis" we mean the process of translating the HDL code to a gate level description, however it might include different design stages. The following steps summarize a full logic synthesis process [27]:

- Write a RTL description of the desired hardware functionality;
- Write a testbench to test the design description (VHDL or Verilog are typical languages for this task as well);
- Verify the functionality of the design with the testbench;
- Phisical synthesize the design to a gate level;
- Evaluate the gate level description with the testbench;
- Verify the post synthesis time requirements are met.

Most FPGA projects are standalone systems interconnected to the outer world. Therefore a large part of debugging and test is concerned with simulating everything will interact with the FPGA. Such process is called the testbench. It simulates feeding signals for the FPGA and it observes the transformations and translations of signals as they propagate through the FPGA from the input pins until they eventually reach the output pins.

The synthesizer can gain information about the design to be implemented in two ways called inference and instantiation. Inference is the ability of getting all the required design functionalities from the HDL. Instantiation instead uses pre-defined structures selected by the user. Clearly the first method is more portable since all the information related to the targeting hardware are provided by the tool and are not dependent by the FPGA fabric resources. Instead instantiation has the benefit of being more optimized as placement, routing and performance are completely predictable. An example of instantiation is the use of an Intellectual Property (IP) core (a standalone configurable hardware block that is instantiated as a black box).

Place and route

In Section 3.1 we have seen that the basic building block of an FPGA is the CLB that usually contains a small number of registers and LUTs. The first step during the place and route compilation phase is the mapping of the obtained netlist to a set of these basic logic blocks. If a primitive gate has more inputs than a single

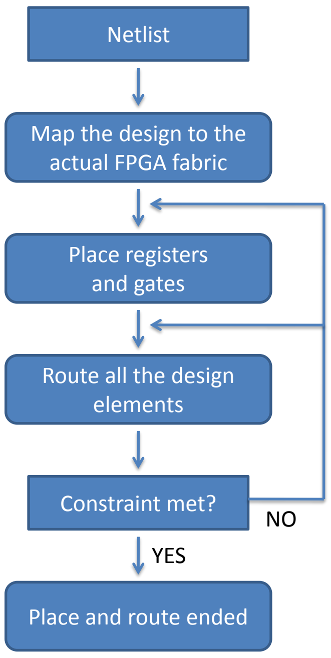

Netlist
Map the design to the actual FPGA fabric
Place registers and gates
Route all the design elements
Constraint met?
NO
YES
Place and route ended

Figure 3.7: Place and route diagram flow

LUT its functionality must be spread across several of them. On the contrary if the primitive gates have few inputs they can be clustered together in a single LUT.

In Fig. 3.7 the entire flow diagram of the place and route process is reported. Once the mapping is complete, the place and route process is charged to decide where to place the registers and gates within the FPGA fabric and to determine the best path for connecting them. The process is based on a series of iterative selection of the fabric cells, their location and the interconnect routing paths. Since randomization is at the basis of the place and route algorithm, especially at the beginning of the placing there is a not negligible probability that related logic is not placed together. In order to overcome this problem the design should be implemented with hierarchy. Indeed if the hardware blocks of the project are well structured and encapsulated one inside the other the place and route process can gain useful information for starting the placement from scratch. A static timing analysis phase comes immediately after the conclusion of the routing. If some of the given constraints are not met the algorithm re-starts the placing or routing phases.

The placing algorithms adopted by the majority of FPGA manufactures are based on annealing [92]. Annealing algorithms for placement are characterized by the their acceptance of a percentage of high-cost permutations of logic blocks rather than of only small permutations that, if the starting point is not a already partially arranged framework, may lead to a local minimum. For the routing phase the most used algorithm is based on the the Dijkstra's shortest path and is called mazerouting [70]. Such algorithm starts to route each net sequentially evaluating the cost on the basis of the number of wire segment needed and with the aim of avoiding congested resources.

Bitstream loading

The last phase of the compilation process of an FPGA is the loading of the configuration instructions to the hardware device. The collection of all the configuration data (including LUTs, CLB, switches, etc.) is called bitstream. The term comes from the most used technique for deploying this data on the FPGA a serial stream of bits sent by an external flash memory chip, although modern FPGAs adopt parallel methods in order to speed up the loading phase. Dedicated internal memories can be also adopted to store the configuration data decreasing the configuration time as well. The need of a speeding up in the deployment of a bitstream in FPGA is mainly due to new technique of partial run-time reconfiguration [29]. Instead of reconfiguring the whole FPGA, a partial bitstream can configure only a part of the chip while the other part keeps running.

3.3 High level synthesis

The traditional hardware design flow usually begins with a high level application description, goes through a Register Transfer Level (RTL) model and ends in a gate-level netlist that can be directly mapped into hardware. While the second translation (from the RTL model to the gate-level specification) is commonly taken by a synthesizer there is still no standard practice for the first translation. HDLs, such as VHDL [109] and Verilog [108], are a well proven and established standard for hardware design, but force the designer of ASICs to think at the RTL level for which HDLs are the perfect match. In other words, HDLs are characterized by a low level of abstraction: HDL is for hardware what assembly is for software.

The silicon process of integration is evolving at an accelerated pace and the consequently complexity growth of the circuits forces a transition from the usual low-level HDLs to languages with a higher level of abstraction. In order to have a good time to market, nowadays, the use of a HLL for hardware synthesis has became mandatory. HLLs obviously ease the circuit design giving more flexibility, but on the other hand they drift away from the best design synthesis quality achievable with a low-level description [30]. We can count many attempts to give a fine grained level of customization to a HLL that resulted in as many tools for hardware synthesis. For example we mention the addition of hardware related constructs to the language or compiler directives that can drive the synthesis process. The main issue that concerns all these tools is the trade off between the use of hardware related structure on one side and the loose of full control of the system being implemented on the other. In this section we will present some of the most famous HLL and tools adopted for the hardware synthesis.

3.3.1 Early HLLs for hardware synthesis

The fist language we encounter moving higher from the RTL is the behavioral part of VHDL. We have given a brief review of the language in Section 3.2.3. In particular Example 3.2.2 and 3.2.3 highlight the difference between a structural and behavioral design. The high-level constructs that can be used in behavioral description are very useful for the hardware description, but they are not always synthesizable. They can be used in a testbench for simulation and verification, but the synthesizer cannot derive automatically a layout from them. Another disadvantage of behavioral synthesis is that, even if it can lead to a synthesis, the quality of the result could be rather poor and unpredictable due to the fact that the same code can represent different hardware circuits.

Example 3.3.1. While structural VHDL is always fully synthesizable, a behavioral description can contain some statements that cannot be translated in hardware design. A typical example is the `wait` statement: it is used to hold a process for a fixed number of clock cycle or until an event occurs. In Figure 3.8 a non-synthesizable behavioral process is reported: it describes the D flip-flop presented in a synthesizable version in Example 3.2.2.

Figure 3.8: Wait statement in a behavioral VHDL process

```
process(CLK, DATA)
begin
  wait until CLK='1';   // wait until condition
  Q <= DATA;
end process;
```

The first attempts to move from an extended low-level HDL to a hardware generation based on an algorithmic description can be classified as domain-specific. Cathedral [24], for instance, was a tool used for digital signal processing (DSP) specifications. It employs a pure behavioral language without any structural construct called Silage. The reason why this languages did not take off is that the domain specialization was not appropriate for the majority of ASIC designs and because of its weak results compared with the one obtained by a manual schematic approach (still in use at the time of investigation).

3.3.2 Advances in HLLs for hardware synthesis

Many high level environments have been proposed to unify the hardware engineers and the software developers through the use of a common language mainly based on imperative languages. Nevertheless the inherent lack of concurrency and timing (essential elements for the hardware synthesis) of such languages made none of them a standard [39]. On the other hand extensions of commonly adopted hardware description languages (HDLs), like SystemVerilog [103], still require to the programmers to own a strong hardware background and they are too specific to be used as general purpose languages.

In this Section we will give a non-exhaustive overview of the most famous C-like languages suitable for hardware synthesis. For more details we suggest [21] and [30]. Even if this thesis is more related with declarative programming such a review is mandatory since we must admit that the most of the languages used by hardware engineers are imperative and we can count only few attempts to ease the hardware synthesis from a non-imperative point of view. We will discuss them later at the end of this Section.

C-like languages

First language that rises the level of abstraction over the usual RTL is Cones [99]. It made use of a very limited subset of C: only few constructs like `if`, `switch` or `for` are supported. Macros are also used to represent array of signals or temporary variable, while the memory elements should be controlled manually without the possibility of employing controllers or schedulers. Even if the resulting design is very poor due to a lack in synthesizing sequential circuits, the EDA tool based on Cones is the first one to generate a gate-level design starting from a real behavioral description.

The main problem common to this language and to the majority of all the C-like implementations is the uncertainty in the final hardware design obtained by the synthesis [39]. Indeed the behavior of a high-level statement can correspond to many different gate-level implementations. HardwareC [65], SpecC [34] and Handel-C [55] are instances of HLLs that adopts low-level hardware constructs to aid the compilation process. HardwareC addresses the problem of parallelism allowing the programmer to use concurrent processes that exchange data through message passing. Furthermore it allows hardware constructs in the code like time and spatial constraints and the possibility of designing with hierarchical structures. SpecC adds to these features the constructs needed to specify finite state machines (FSMs) and pipelining. It is developed specifically for the synthesis of integrated systems: the typical architecture is a processor based system with addition of custom hardware. SpecC can be considered an ancestor of the system level languages that will be introduced in 6.1. Finally a HLL specifically designed for FPGAs and ASICs is HandelC. It adds to C explicit parallelism by using keywords that point out instructions that has to be executed in parallel by the different parts of the hardware circuits. Channel communication is responsible for exchanging data between parallel processes and for synchronizing them. Its semantics is, indeed, derived from the former formalism Communicating Sequential Processor [59] with the limitation that a variable cannot be written by different parallel processes, but can only be read. Moreover it lacks features typical of general purpose processor languages like pointers or floating point arithmetics.

All the languages reported in the last paragraph have the advantage over Cone to be able of synthesizing more complex and advanced sequential circuits, but each of them has as drawback the need of low-level constructs to drive the synthesis tool. Such hardware related parts of the code make these language oriented to hardware engineers rather than software programmer. However the reason why these tools never took off is to be searched in the overhead introduced by the hardware constructs. Basically the C specification has to be rewritten, in term of hardware constructs, in order to met the synthesis requirements, resulting in a double work for the programmers. Moreover the re-usability of the code is very limited since the added hardware constructs are device dependent.

Another way of integrating hardware directives in the high-level synthesis process is the use of configuration files. Spark [54] is an instance of such HLLs. It accepts as input behavioral C code, without pointers and function recursions, and produces synthesizable structural VHDL that is standardly synthesized by commercial tools. During a pre-synthesis phase it employs parameters taken from the configuration files to tune heuristics through function inlining, loop unrolling, loop fusion, common sub-expression elimination, loop-invariant code motion, induction variable analysis and operation strength reduction. Following a scheduler phase performs compiler transformations as percolation, speculative code motion and changing across conditions. Also the resource allocation has to be done by the programmer via the configuration file where it is possible to specify the available hardware resources and the timing constraints. The final stage of Spark compilation is the control generation: a FSM that implements the controller.

Analogue methodologies are used by HLLs as C2Verilog [98], Cyberg WorkBench [31] and AutoPilot [11]. They need a configuration file that contains general requirements of the programmer compared to the low-level hardware constructs. The main disadvantage of these languages is still the poor result due to the inadequate internal optimization framework. The communication to the tool is limited to the tuning of the predefined heuristics and is not possible to perform fine-grained actions like to specify which loop to parallelize. In order to address this issue another class of HLLs is developed. Their synthesis flow is aided by compiler directives (pragmas). In such a way a more fine-grained optimization is achievable, still maintaining the re-usability of the code. Moreover writing a pragma instead of changing the structure of the code is less error prone than to adopt a HLL based on hardware constructs. The most well-known instance of HLLs with compiler directives is ImpulseC [62]. Such a language lets the designer make use of pragmas for, as example, informing the compiler that a particular loop can be pipelined or a given array cannot be scalarized. ImpulseC is mainly used for the synthesis of hardware accelerator based on FPGAs. The architecture is generated by the communicating sequential process described in C language and is completely dataflow oriented: processes accept data via shared memory and send them back.

Among the synthesizers that accept compiler directives, we can cite Catapult C [23] by Mentor Graphics or the academic ClooGVHDL [33]. The compilers use the pragmas to be driven during the synthesis that can be ignored in case of standard compilation on a general purpose processor. However since the hardware compilation require particular constructs the code should be rewritten if performances are needed. Anyway the software compilation is useful in case of simulation or verification of the hardware behavior.

A very important class of HLLs for hardware synthesis is the one that adopts specific class libraries that extend standard C in order to meet the hardware description requirements. The most popular framework of such category is SystemC [102]. It is based on C++ adding important concepts as concurrency (of multiple processes), time events and data types. Constructs that recall hardware concepts like modules, ports and signals are provided as class library as well as concurrent processes, statements for waiting a clock edge or hardware data types (bit or bit vector). SystemC can be seen as a methodology for describing software algorithms and hardware architecture but also system level design [83] (in Chapter 6 we will discuss this later aspect). SystemC has semantic similarities to VHDL and Verilog, but it has a syntactical overhead, compared with these languages, when used as a HDL because it is intended mainly for specification, architectural analysis, testbenches, and behavioral design. Furthermore skepticism about the usefulness of C++ design flow is expressed in [51] where the concern about the rising gap between the models and the synthesis is pointed out. Especially block-level designers claim that C++ is not the right direction for HDL development because synthesis and verification impose much stronger requirements on the language.

Example 3.3.2. In this example we want to provide an example of SystemC code used for modeling synchronous logic. In Figure 3.9 a basic model for a D flip-flop is reported. Please refer to Example 3.2.2 and 3.2.4 for a comparison with VHDL.

Figure 3.9: SystemC specification of a D flip-flop

```
// File: D_ff.h
#include "systemc.h"

SC_MODULE (D_ff) {
  sc_in<bool> data, clk;
  sc_out<bool> q;

  void prc_D_ff();

  SC_CTOR (D_ff) {
    SC_METHOD (prc_D_ff};
    sensitive_pos << clk;
  }
};
```

```
// File: D_ff.cpp
#include "D_ff.h"

void D_ff::prc_D_ff () {
  q=data;
}
```

The D flip-flop description consists of two files: the header file contains the module description and the process declaration while the C++ program defines the process. After the inclusion of the SystemC class the module is described by a port declaration in a very similar way of an entity in VHDL. `SC_CTOR` declares the process of type `SC_METHOD` that is sensitive to a list of signals (sensitivity list in VHDL). In the current example the process is sensitive to the positive edge of the clock `clk`. The C++ file specifies the action taken by the process when it is activated: actually the copy of `data` to `q`.

Other paradigms

Among the functional languages we can count a large number of successful approaches to hardware description. Since the 80s one of the most popular domains in which functional languages have been extensively used is hardware design [93]. General purpose functional languages, like Haskell, have been widely used as host languages for embedding HDL [18, 69]. Other examples of declarative hardware oriented languages are Pebble [74] or Ruby [63] that support structural descriptions based on abstractions such as blocks and their interconnections. They allow the user to focus on the essential structure of the system describing parameterised design concisely thanks to features such as iterative descriptions and static recursion in the circuit design. These extensions provide simple meta-languages that helps programmers to deal with complex circuits rather than using a poor structural HDL.

Another remarkable approach to the introduction of a new level of abstraction, developed in order to fulfill the gap between system specification and interface to implementation, is constituted by property specification languages. In fact they allow hardware synthesis from a formal specification (very close to natural language) that is given by mathematical operators. In such a way the functional verification step is eliminated by the classical design flow because the generated HDL code is correct by construction. The pioneer works on developing control-type sequential circuits [42, 91] use respectively free and reduced-ordered Binary Decision Diagram to support the synthesis process, but these kinds of algorithms lead to the state explosion problem which compels the applications to a restrict class of simple designs. More recent methods take as input specification standard assertion language. Bloem [19], with an automata based approach, developed a polynomial algorithm starting from a subset of the Property Specification Language (PSL) [44]. More recently a tool proposed by Borrione in [81], using a modular construction, generates in linear time circuits that comply with the PSL semantics.

Logic programming and especially Prolog [28] are used as formalisms for hardware design specification and verification as well. In [107, 101] it was initially shown how logic programming can be used as an efficient design tool respectively for sequential and concurrent system. Further work [35] illustrates how the essential requirements of a HDL are satisfied using fundamental features of Flat Concurrent Prolog and how it can overcome known disadvantages of common HDL like overloading, verbosity or the lack of composite `if` statement. More recent approaches [13, 14] present a Prolog-based hardware design environment based on a high-level structural language called HIDE. Such language was developed with the precise purpose of filling the gap of the structural HDL languages that can only deal with small circuits. Indeed the HDL description tends to be very complex due to the need of making all the connections explicit. The language finally evolved in HIDE+ that is more suited for describing parallel irregular architecture and provides a more robust control mechanism that allows for complex clocking schemes.

HIDE uses the base notation of Prolog for composing the hardware design starting from a simple and small set of elementary hardware blocks. Indeed HIDE is a pure structural language: it assembles compound blocks from primitives. Each block can be defined in a two or three dimensional plan and is specified by its position in the array and by the connections with the neighboring blocks. Each connection is composed of ports with directions (in or out) and controls that are a special kind of ports used to broadcast signals to a large number of blocks. Circuits can be combined by blocks to create higher level blocks hierarchically, using several constructors that also help to replicate the blocks.

Example 3.3.3. To illustrate HIDE language we report in Figure 3.10 an example of an implementation of an eight-bits adder. It is based on a vertical composition in a two dimensional array of the elementary block two-bits full adder.

Figure 3.10: HIDE specification of eight-bits adder

```
is basic block( add2, 1, 1, [clk],
[ [(co,output)], [(ci,input)],
[(a1,input), (b1,input), (a2,input), (b2,input)],
[(s1,input), (s2,input)],
[],[] ])

Adder8 = v_seq(4, add2)
```

The first line introduces the Prolog fact that represents the two-bits full adder named `add2` and that occupies one block of the array in vertical and one in horizontal and, finally, it has the clock signal as control input. The `add2` component has two two-bits signals as input on the west side (from the block of the array on its left) `a1,b1` and `a2,b2` and one two-bits output as sum on the east side `s1,s2`. `ci` and `co` are the input and output carries coming respectively from south and to the north side. The last line states how construct the eight-bits adder from the sub-blocks `add2`. The single two-bits full adder is replicated four times and they are disposed in a vertical composition (the south ports of the top block are connected with the north ports of the block immediately down).

II

From CHR to hardware

4 Hardware Compilation

Here we want to face the problem of translating a high-level language as CHR to a low-level hardware specification. Hence the main goal pursued in this Chapter is to reach a RTL level hardware description in a single compilation step starting from a constraint based specification.

At first we investigate in Section 4.1 the features of CHR that could hamper the hardware synthesis, then we address the correspondences between CHR rules and hardware. The main ideas behind our CHR-based hardware specification method will be discussed in Section 4.2. A detailed description of each translation step is given in order to fully specify the mapping between CHR and hardware blocks. In the rest of the thesis we observe the guidelines reported in such Section to implement and execute any translation of CHR rules into VHDL behavioral models of hardware modules, which directly manipulate constraints, and which can be synthesized in a specific technology using existing logic-level synthesis tools. As depicted in Figure 4.1 the complete compilation flow starts from a subset of CHR and goes to implementation on FPGA passing through the low level VHDL language. The second step of compilation is then performed by the synthesizer within the standard compilation flow described in 3.2.1.

The Euclid's algorithm for finding the greatest common divisor is used as a running example of hardware translation along all the Chapter. Section 4.3 is devoted to show the fine-grained design achievable when CHR is used for a structural description. In Section 4.4 we report the outcomes of experimental hardware executions of a CHR specification from the query to the result. Besides the running example, here we present the result of a hardware translation of the Floyd-Warshall algorithm. Finally, in Section 4.5, we analyze the hardware implementation of another typical CHR case study: the merge sort algorithm. In such example we will make use of a simple transformation in the CHR program to allow the source code to be compliant with the CHR subset needed by the hardware compilation. Experimental results of merge sort algorithm are postponed to Section 5.4 were different degrees of parallelization are also discussed.

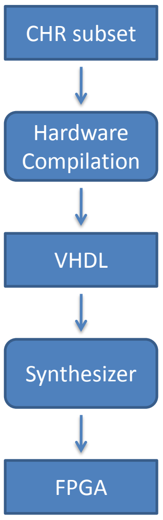


Figure 4.1: CHR to hardware compilation diagram flow

4.1 The CHR subset

Since the hardware resources can be allocated only at compile time (dynamic allocation is not allowed in hardware due to physical bounds), we need to know the largest number of constraints that should be kept in the constraint store. It is not trivial to foresee the maximum number of constraints to be stored during computation. Thus in order to establish a upper bound to the growth of constraints, we consider a subset of CHR, which does not include propagation rules. Programs are composed of simpagation rules of the form:

$$\begin{gathered} rule\,@\,c_1(X_1),...,c_p(X_p)\backslash c_{p+1}(X_{p+1}),...,c_n(X_n) \Leftrightarrow \qquad (4.1) \\ g(X_1,...,X_n)\,|\,Z_1\,is\,f_1(X_1,...,X_n),...,Z_m\,is\,f_m(X_1,...,X_n),c_{i_1}(Z_1),...,c_{i_m}(Z_m). \end{gathered}$$

where X_i $(i \in \{1,\ldots,n\})$ can be a set of variables and the number of body constraints is less than or equal to the number of constraints removed from the head $(m \leq n-p)$ and no new type of constraints is introduced: $\{i_1,\ldots,i_m\} \subseteq \{p+1,...,n\}$. In this way the number of constraints cannot increase and the constraint store can be bounded by the width of the initial query. In Section 4.5 a simple technique used to rewrite simplification rules in order to avoid the introduction of new type of constraints will be shown.

The proposed translation methods assume a ground CHR rule system. Constraints cannot contain free variables and any derivation starting from such constraints can only lead to other ground constraints. Hence the variables present in the right hand side constraints are immediately bounded by the unification with the goal constraints. All the CHR examples provided until now satisfy this condition, but in Section 6.2.4 we will present a non-ground example in which our technique can still be applied because the free variables are never bounded during the full computation and hence they can be substituted by constant values before compilation.

We defer the discussion on the limitations imposed by the adoption of such subset of CHR to Chapter 6 where a hybrid software/hardware execution is proposed in order to bypass these restrictions. In the remaining of the Chapter we consider rules as typed in their *head normal form* (see 2.2): all the arguments in the head of the rule are variables and variables never occur more than once (all equality guards implicitly present in the head are written explicitly in the guard).

Example 4.1.1. Referring to the GCD algorithm reported in Example 2.2.1, it is clear that for such program the number of constraints remains bounded during the computation. Indeed the first rule, if applied, removes a constraint from the store, instead the second removes a constraint and adds a new one, thus leaving the total number of constraints unchanged.

4.2 Principles of the hardware blocks

The framework we propose is logically divided into two parts:

- Several hardware blocks representing the rewriting procedure expressed by the program rules.
- An interconnection scheme among the blocks specific for a particular query.

The first one builds the hardware needed to compute the concurrent processes expressed by the CHR rules of the program, while the second one is intended for reproducing the query/solution mechanism typical of constraint programming.

As depicted in Fig. 4.2 we call Program Hardware Block (PHB) a collection of Rule Hardware Blocks (RHBs), in turn generated by each rule of the CHR program. The proposed approach considers the constraints as hardware signals and the arguments as the values that signals can assume. The initial query can be directly placed in the constraint store from which several instances of the PHB concurrently retrieve the constraints working on separate parts of the store and after computation they replace the input constraints with the new ones. A Combinatorial Switch (CS) sorts and assigns the constraints to the PHBs taking care of mixing the constraints in order to let the rules be executed on the entire store. The following paragraphs explain in details the construction of the blocks.

4.2.1 Rule Hardware Blocks

The hardware corresponding to the CHR rule (Eq. 4.1) has as inputs n signals that have the value of the variables $X_1 ... X_n$ (all the arguments of the head constraints). If $X_1 ... X_n$ are sets of variables we use vectors of signals (*records* in VHDL). The

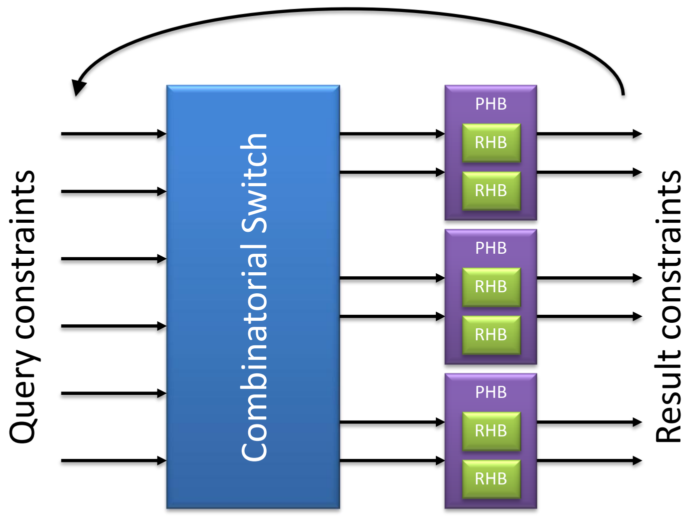


Figure 4.2: Complete hardware design scheme

computational part of the RHB is given by the functions $f_1 \ldots f_m$ that operate on the inputs and the resulting output signals have the value of the variables $X_1 \ldots X_p$ and $Z_1 \ldots Z_m$. Without loss of generality we have assumed that the constraints have single arguments, but a straightforward generalization is possible using vectors of signals (*records* in VHDL). The reason of reassigning a value as output also to the arguments of the kept constraints comes from the need of retaining unchanged the value of their variables. Indeed we do not use an addressable memory (even if it should be possible) to store the constraints and hence the constraint store should be refreshed every computational cycle with the values of the output constraints of the RHBs.

We exploit *processes*, the basic VHDL concurrent statement, to translate the computational part of a rule to a sequential execution. Each rule is mapped in a single clocked *process* containing an *if* statement over the guard variables.

In order to take into account the possibility of a reduction of the number of constraints during the computation, each output signal for a given constraint is coupled with a *valid* signal that states to the following components whether to ignore the signal related to such constraint or not.

Example 4.2.1. Figure 4.3 sketches the RHBs resulting from the two rules of the `gcd` program introduced in Example 2.2.1 and reported here as reference:

```
R0 @   gcd(N) <=> N = 0 | true.
R1 @   gcd(N) \ gcd(M) <=> M>=N | Z is M-N, gcd(Z).
```

Notice that each constraint is associated with two signals: one contains the value of the variable of the constraint (the solid line), and the other one models its validity (dashed line).

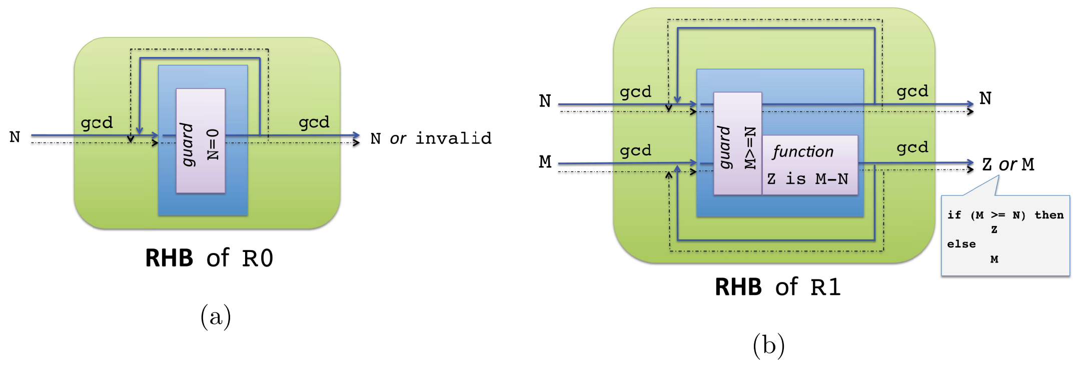


Figure 4.3: The Rule Hardware Blocks for the `gcd` rules.

The block in Fig. 4.3(a), that corresponds to `R0`, has as input, the value for variable `N` together with its *valid* signal. It performs a check over the guard and if the guard holds the *valid* signal is set to false whereas the value of the gcd signal is left unchanged. This simulates at the hardware level the removal of a constraint from the constraint store.

The block in Fig. 4.3(b) is the translation of the second rule of `gcd`. It consists of four input signals, i.e. the values for the variables `N` and `M` with their *valid* signals. In this case the *valid* signals remain unchanged. If the guard holds the signal value of the second input constraint is replaced with `Z = M-N` while the value of the first one is not modified. If the guard does not hold the outputs of the block coincide with the inputs. The computational part is carried out by the subtraction operator.

4.2.2 Program Hardware Block

The PHB is the gluing hardware for the RHBs: it executes all the rules of the CHR program and hence it contains all the associated RHBs. PHB takes as input the two global input signals *clk* and *reset* used for synchronizing and initializing purposes. It provides for the *finish* control signal used to denote when the outputs are ready to be read by the following hardware blocks. The RHBs keep on applying the rule they stand for until the output remains unchanged for two consecutive clock cycles.

Note that in the hardware each constraint is represented as a different signal. If the head of a rule contains more than one constraint of the same type, the corresponding signals must be considered as input in any possible order by a RHB encoding the rule. This is obtained by replicating RHB a number of times equal to the possible permutations of the constraints of the same type. Finally we have to guarantee that only one copy of the RHB can execute per clock cycle.

More precisely, taking into account the generic rule 4.1, we can assemble the head constraint c_i into different groups $g_1, ..., g_q$ depending on the type of the constraint. Calling $k_1, ..., k_q$ the number of constraints belonging to a each group ($\sum_{i=1}^{q} k_i = n$ is the total number of inputs of a RHB) we can compute the total number R of

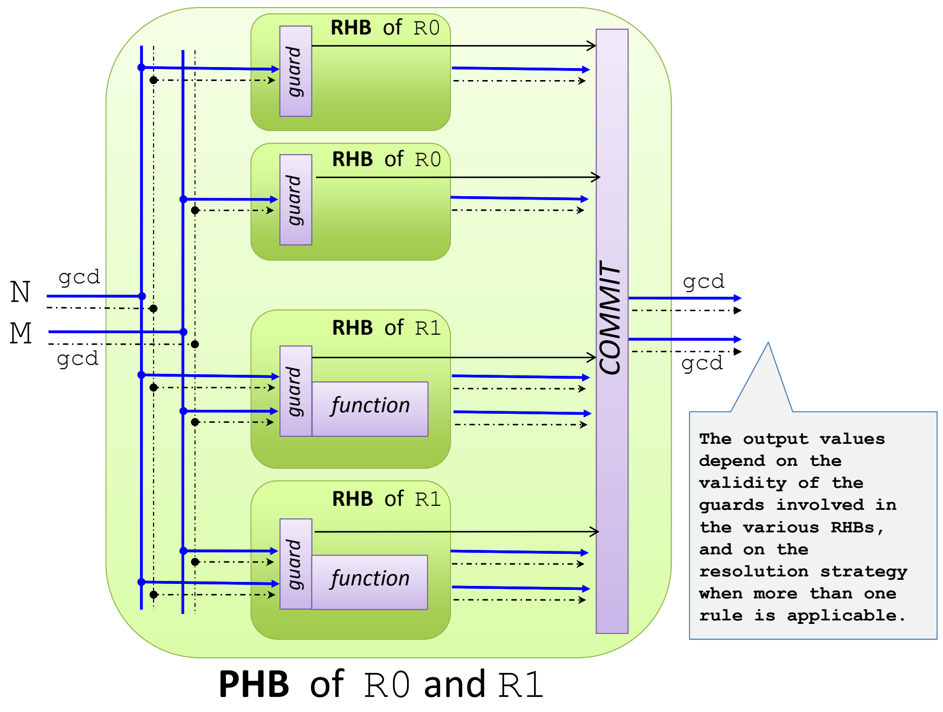

RHB of R0
guard
RHB of R0
guard
N
gcd
M
gcd
RHB of R1
guard
function
RHB of R1
guard
function
COMMIT
gcd
gcd
The output values depend on the validity of the guards involved in the various RHBs, and on the resolution strategy when more than one rule is applicable.
PHB of R0 and R1

Figure 4.4: The Program Hardware Block for the `gcd` program

RHBs needed as:

$$R = \prod_{i=1}^{m} k_i! \tag{4.2}$$

Example 4.2.2. Let us consider rule `R1` described in Example 2.2.1. Two instances of `gcd` are present in the head of the rule and hence two signals are created respectively with value `N` and `M` and they are the inputs of the RHB. Due to the guard of `R1` these inputs feed a comparator that checks if the value of the second signal is greater than or equal to the first. If the condition is satisfied the value of the second signal is replaced by the result of the subtractor that has as inputs the two signals.

According to what stated in Equation 4.2 the total number of RHB instances needed in the PHB is two since it has as input only one group of constraints constituted by two constraints of the same type and hence the total number of permutation of its input of the same type is 2!. Indeed let us consider the case in which `N` is greater than `M`, the rule can fire as well because the head constraints are of the same type and so they can be swapped. For this reason PHB has to contain another copy of the RHB that executes such rule but with inputs in reverse order (see Figure 4.4).

The PHB level is also used to set the rules parallelization at the basis of the computation. As we said each rule is executed by one or more concurrent *processes* that fire synchronously every clock cycle. Therefore we exploit the notion of strong parallelism of CHR, introduced in Section 2.2.2, assuming that rules can work on common constraints at the same time if they do not rewrite them. If two rules try to change the same constraint then we cannot parallelize and we need to execute them one after the other if the latter is still applicable. For instance in the PHB of the running example, presented in Example 4.2.2, rule `R0` cannot be executed in parallel with `R1` because they could rewrite the same constraint. The commit block at the output stage is then used to give a priority to the rules execution. According to the theoretical operational semantics [46], we can state that the provided rule application is fair since every rule that could fire does it every clock cycle or in the worst case in the subsequent cycle.

A more detailed description of the parallelism achievable in the PHB will be given in Section 5.3.

4.2.3 Combinatorial Switch

A further level of parallelization is achieved replicating the PHBs into several copies that operate on different parts of the global constraint store (weak parallelism, see Section 2.2.2). PHBs can compute independently and concurrently because they are attempting to rewrite different constraints. Although they process data synchronously, since they share a common clock, it is not required that they terminate computation at the same time. Indeed the CS acts as synchronization barrier letting the faster PHBs wait for the slower ones. It is also in charge to manage communication among hardware blocks exchanging data: once all the PHBs have provided their results, it reassigns the output signals as input for other PHBs guaranteeing that all the combinations between them are covered. Exploiting the fact that the number of constraints cannot increase, the CS works directly on the signals coming from the PHB, but there are no impediments to retrieve the constraints from an external memory if the space capacity of the FPGA is not sufficient.

In practice the implementation of this interconnection element relies on a signal switch that sorts the n query constraints according to all the possible k-combination on n (where k is the number of inputs to the single PHB) and connects them to all the inputs of the PHBs. The maximum number of PHBs that can always work in parallel on the constraint store is $\lfloor n/k \rfloor$, since its width is limited by the width of the query n. Indeed we cannot feed the same signal (and hence the same constraint) to different inputs of the PHBs (to match with more than one head constraint) according to weak parallelism.

Implementing CS as a finite state machine leads to a total number of states S equal to the number of possible combinations divided by the number of concurrent PHBs:

$$S = \frac{\binom{n}{k}}{\lfloor n/k \rfloor} \approx \frac{\prod_{i=1}^{k-1} n - i}{(k-1)!} \tag{4.3}$$

Despite a good degree of parallelization achieved by the CS (it makes $\lfloor n/k \rfloor$ PHBs try to execute in parallel as many rules), it needs a number of states $O(n^{k-1})$

Table 4.1: Gcd derivation in hardware

	PHB_1	PHB_2	PHB_3
Reset	$\{gcd(6), v\}, \{gcd(12), v\}$	$\{gcd(45), v\}, \{gcd(15), v\}$	$\{(gcd(9), v\}, \{gcd(33), v\}$
R1 R1 R1	$\{gcd(6), v\}, \{gcd(6), v\}$	$\{gcd(30), v\}, \{gcd(15), v\}$	$\{gcd(9), v\}, \{gcd(27), v\}$
R0//R1 R1 R1	$\{gcd(0), n\}, \{gcd(6), v\}$	$\{gcd(15), v\}, \{gcd(15), v\}$	$\{gcd(9), v\}, \{gcd(18), v\}$
$\emptyset$ R0//R1 R1		$\{gcd(0), n\}, \{gcd(15), v\}$	$\{gcd(9), v\}, \{gcd(9), v\}$
$\emptyset$ $\emptyset$ R0//R1			$\{gcd(0), n\}, \{gcd(9), v\}$
$CS\{(1,3),(2,5),(4,6)\}$	$\{gcd(0), n\}, \{gcd(0), n\}$	$\{gcd(6), v\}, \{gcd(0), n\}$	$\{gcd(15), v\}, \{gcd(9), v\}$
$\emptyset$ $\emptyset$ R0//R1			$\{gcd(6), v\}, \{gcd(9), v\}$
$\emptyset$ $\emptyset$ R0//R1			$\{gcd(6), v\}, \{gcd(3), v\}$
$\emptyset$ $\emptyset$ R0//R1			$\{gcd(3), v\}, \{gcd(3), v\}$
$\emptyset$ $\emptyset$ R0//R1			$\{gcd(0), n\}, \{gcd(3), v\}$
$CS\{(1,4),(3,5),(2,6)\}$	$\{gcd(0), n\}$, {gcd(0),n}	$\{gcd(6), v\}, \{gcd(0), n\}$	$\{(gcd(0), n\}, \{gcd(3), v\}$
$CS\{(1,5),(2,4),(3,6)\}$	$\{gcd(0), n\}, \{gcd(0), n\}$	$\{gcd(0), n\}, \{gcd(0), n\}$	$\{gcd(6), v\}, \{gcd(3), v\}$
$\emptyset$ $\emptyset$ R1			$\{gcd(3), v\}, \{gcd(3), v\}$
$\emptyset$ $\emptyset$ R1//R0			$\{gcd(0), n\}, \{gcd(3), v\}$

in order to execute all the possible combinations on the input signals. Since the time necessary for evaluating the query is proportional to the number of states, it is important to limit the number of inputs for each PHB (k). This leak in performance can be considered as the demonstration of the search for matching problem: a very well know issue in CHR [97] and in general in multi-headed constraint languages. In Chapter 5 we will discuss how to improve time complexities to space complexity's cost giving optimized structures for the CS.

Example 4.2.3. Now that the functionality of all the required hardware blocks has been described we are ready to see an example of hardware computation of CHR. Keeping on with the running example 2.2.1 in Table 4.1 the execution of the query:

```
gcd(6), gcd(12), gcd(45), gcd(15), gcd(9), gcd(33).
```

is reported. The full hardware design consists of a CS with six inputs and six outputs and three PHBs (with two inputs) like the ones described in Example 4.2.2.

At reset all the query constraint are read by the CS, they are sorted and in pairs they are passed to the three PHBs. For reference we assign the label 1 and 2 to the outputs of PHB_1, 3 and 4 to PHB_2 and 5 and 6 to PHB_3. To denote the operations of the CS on the signals we use the following notation: $CS\{(\_,\_),(\_,\_),(\_,\_)\}$ where the three couplets $(\_,\_)$ are the ordered inputs to the three PHBs (from the first input to the sixth one) and the value inside parenthesis are the label of the output of the PHBs. For instance $CS\{(1,2),(3,4),(5,6)\}$ means that the first output of PHB_1 is connected to the first input of PHB_1 (since number one occupies the first position of the set), the second to the second and so on and so forth for all the PHBs. Between the swapping of the CS the computation inside each PHB is represented by a tern of rules where each element can assume the value R0 or R1 depending on which rule fires, $\emptyset$ if no rule can be applied or R0//R1 if they both fire in parallel. Indeed, even if they work on the same constraint, actually R0 modifies only the valid signal while R1 modifies only the value of the constraint argument. Thus we refer to a parallel execution because the two rules can be applied in the same clock cycle even if they logically fire one after the other (R0 after R1). Finally each signal is represented as a pair of a constraint and a 1-bit value that returns its status: valid (v) or not-valid (n).

In the first column of Table 4.1 is reported the action performed by the hardware on the constraints that can be a permutation provided by the CS or the application of one or more rules concurrently. In the other three columns the derivation of each PHBs is presented. Their computation is completely independent and the only way for exchanging the constraints is after a CS swapping. Each row of the Table corresponds to a different computational phase triggered by the CS.

Figure 4.5: VHDL translation of rule 4.4 in a D flip-flop

```
architecture DFF_arch of DFF is
begin
  process(clk, d)
  begin
    if clk'event and clk='1' then
      q <= d;
    end if;
  end process;
 end DFF_arch;
```

4.3 A structural implementation

The gcd program presented in Example 2.2.1 shows that we can implement in hardware the Euclid's algorithm at the behavioral level, i.e., it describes a system in terms of what it does rather than in terms of its components and interconnections. Instead, here we illustrate an example of structural hardware design which shows the hardware granularity achievable generating the VHDL code for the basic building block of sequential circuits directly implementable in FPGA. According to the general scheme for rules representation described above, the following CHR rule implements the D flip-flop, the elementary memory block capable of storing the value of a signal:

```
d(X) \ q(_) <=> q(X).        (4.4)
```

where `d/1` and `q/1` stand for the input and the output signals of the D flip-flop. The `q` constraint is rewritten every time a `d` constraint is present, as in the D flip-flop, every clock cycle, the value of the output is replaced by the value of the input.

Such kind of notation is intended for a design representation and not for an algorithmic description. In order to obtain a D flip-flop we should apply just the transformations described in Section 4.2.1, indeed there is no need of the query/result mechanism. The corresponding VHDL code is reported in Figure 4.5 where a positive edge clock system is considered. It is worth noting that a CHR program consisting only in rule 4.4 would have no termination. The removed constraint `q` is then reintroduced by the rule and hence, since `d` is not removed, the rule can be applied again. That is, basically, what hardware does: the flip-flop continuously copy its input signal on the output at every clock cycle.

By using again the idea that the removal and the introduction of the same constraint corresponds to a memory refresh, we can implement more complex circuits. The following table presents the two lines code that describes the hardware circuit of a 2-bits counter represented in Fig. 3.3:

```
      b0(X) <=> Z is (not X), b0(Z).
b0(X) \ b1(Y) <=> Z is (X xor Y), b1(Z).        (4.5)
```

Figure 4.6: VHDL translation of rules 4.5 in a 2-bits counter circuit

```
architecture 2BITS_arch of 2BITS is
begin
  process(clk, b0)
  begin
    if clk'event and clk='1' then
      b0 <= not b0;
    end if;
  end process;
  process(clk, b0, b1)
  begin
    if clk'event and clk='1' then
      b1 <= b0 xor b1;
    end if;
  end process;
 end 2BITS_arch;
```

where `not` and `xor` are operators predefined in HDL (built-in) that are used to implement the combinatorial logic part of the circuit. In Figure 4.6 the VHDL code corresponding to rules 4.5 is reported. Two parallel processes are used to build up the 2-bits counter because each of them derive from a rule, even if the assignment of `b0` and `b1` could be done in a single process. With respect to the code in Figure 3.4 and 3.5 we should notice that the level of abstraction used in such VHDL description is not behavioral, but it does not look like the structural even if it is fully equivalent. Indeed, for simplicity, we have loose the hierarchy block construction, but the code still remains structural and the compilation of the two description will result in the same hardware.

4.4 Experimental results

We use the automatically generated hardware blocks described in Section 4.2 to implement in FPGA two CHR programs. Initially we resume with the running example and afterwards we will show the outcome implementation of the Floyd-Warshall algorithm for finding shortest paths in a weighted graph. Part of the produced VHDL code is reported in Appendix A. It is worth noting the number of code lines needed in VHDL with respect to CHR. The latter approach clearly results to be time saving for the programmers.

4.4.1 Greatest common divisor

Here we describe the hardware implementation of the algorithm presented in Example 2.2.1 for finding the greatest common divisor at most of 128 integers.The

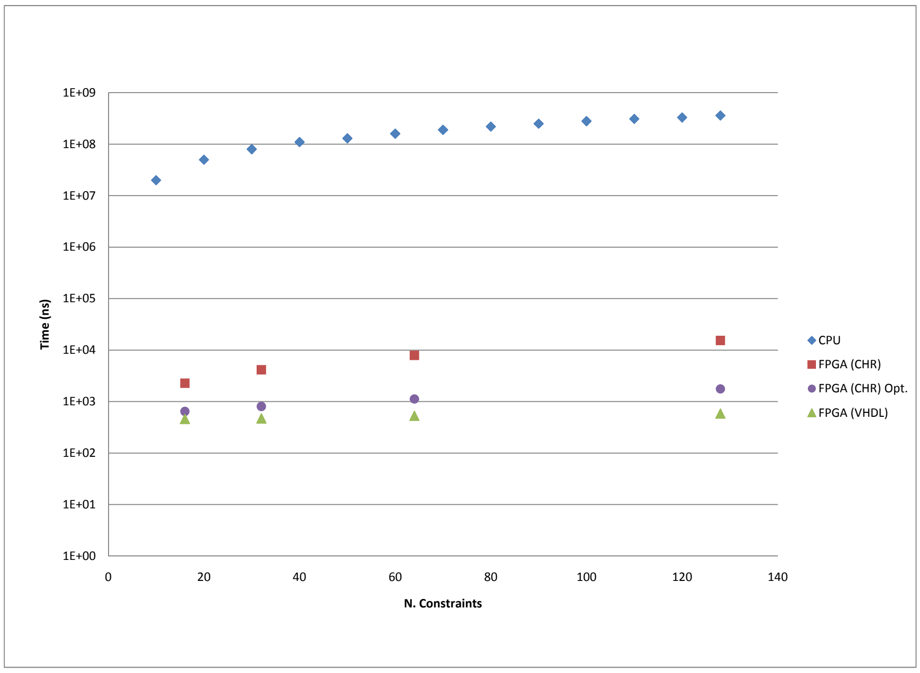


Figure 4.7: Gcd execution time (log scale)

resulting hardware design relies on 64 PHBs deriving in parallel the gcd while the CS pairs the constraints in a round robin tournament scheme where each constraint is coupled once with each other. For comparison purposes we implement the same algorithm directly in behavioral VHDL using a parallel reduction that exploits the associative property of the gcd operation (basically a knockout system where we compute the gcd in parallel of 64 pairs then of 32 and so on). Both hardware specifications are then synthesized and simulated with ISim the Xilinx ISE simulator at 100MHz reference clock frequency. Figure 4.7 reports the execution times for 16, 32, 64 and 128 1-byte integers. The two FPGA implementations are labeled respectively as FPGA (CHR) and FPGA (VHDL). The curve labeled CPU refers to the computational time of the CHR `gcd` program running on Intel Xeon 3.60GHz processor with 3GB of memory. It is displayed just for an order of magnitude reference since we cannot compare them due to the completely different hardware nature.

The plot clearly shows how the execution time can increase to more than an order of magnitude with respect to the VHDL solution. This is primarily due to the fact that the Combinatorial Switch does not take into account that the number of constraints can decrease, and hence the number of possible combinations. In Section 5.1 we address this issue suggesting an optimisation for rules that have the property of strong parallelism like in the gcd case. The outcome of such optimisation is also reported in Fig. 4.7 (labelled as FPGA (CHR) Opt.) and it exhibits a relevant reduction of the execution time. Notice that the VHDL implementation leads to

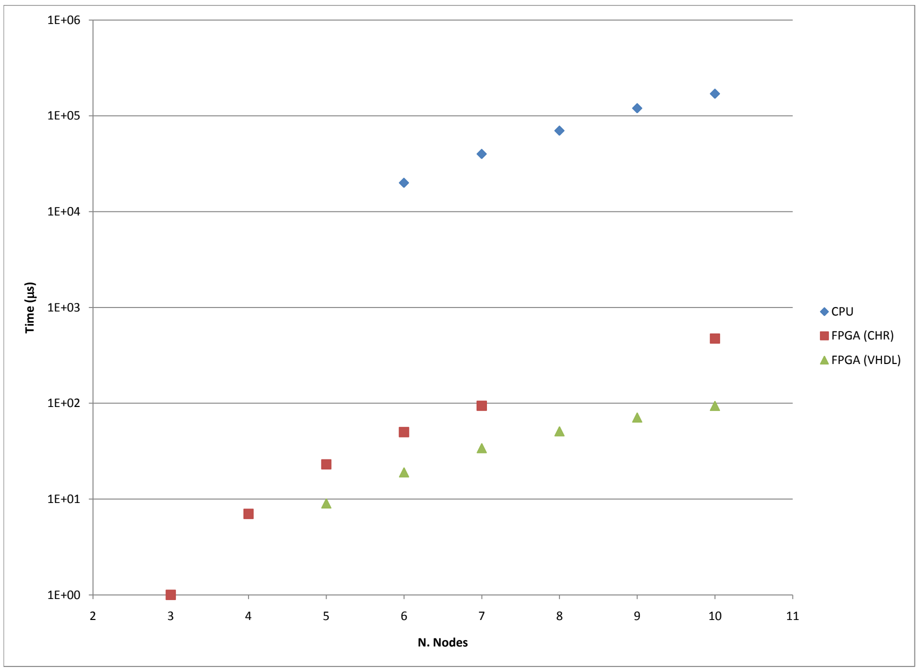


Figure 4.8: Floyd-Warshall execution time (log scale)

an execution time almost constant due to the complete parallelism achievable by hardware. We do not observe a super-linear trend in our implementation like the one noticed in [100] because the derivations of each pair of `gcd` constraints do not interfere each other. Finally we should notice that the resulting highest frequencies of operation are all above 250 MHz and up to 350 MHz, which is quite good for a non pipelined architecture.

4.4.2 Floyd-Warshall algorithm

Similar results are obtained on a different example, namely the implementation of Floyd-Warshall algorithm for finding the length of the shortest paths between all pairs of vertices in a weighted graph. A procedural version of the algorithm follows:

```
1  for k=1 to N
2     for i=1 to N
3        for j=1 to N
4           $d_{i,j}$=min($d_{i,j}$,$d_{i,k}$+$d_{k,j}$)
```

where $d_{i,j}$ are the elements of the matrix representing the graph. In CHR the algorithm can be expressed as a simple rule with three constraints in the head standing for three edges that should be taken into account for the minimum computation:

```
edge(I,K,D1), edge(K,J,D2) \ edge(I,J,D3) <=>
                  D3>D1+D2 | D4 is D1+D2, edge(I,J,D4).          (4.6)
```

From such rule our method generates a simple PHB that has as inputs and outputs three triples of signals that are respectively: the source, the destination and the weight of the edges. Depending on the query dimension n a CS with $\frac{1}{2}(n-1)(n-2)$ states assigns the constraints to $\lfloor \frac{n}{3} \rfloor$ PHBs. In Fig. 4.8 we compare our implementation with the VHDL based one described in [77] derived by a logic-level specification. As we can see from the plot, our implementation exceeds the best handcrafted design only by less than one order of magnitude and at the same time it delivers a high degree of flexibility: you can manipulate a one-line code rather than rearrange a fixed architecture of hundreds of lines.

4.5 Merge sort implementation

In this Section we want to trace and summarize all the translation steps applied to the running example in this Chapter. As candidate for a complete hardware implementation we choose a more interesting CHR program implementing the merge sort algorithm in optimal complexity [46]. The upper part of Figure 4.9 reports the rules that sort a sequence of m numbers N_m, where m is a power of two. The numbers N_m should be encoded in the query as constraints $seq(1, N_x)$ with $1 \leq x \leq m$, and where 1 is the initial length of the sequence. Rule `M1` initializes the `arc/2` constraints (arcs between numbers whose union constitute a sequence) for the actual merging done by rule `M0`. `M1` prepares arcs for `M0` starting from sequences of the same length and produces new sequences with the length increased by one.

In order to understand whether the two rules belong to the CHR subset implementable in hardware, we should notice that `M0` leaves unchanged the number of `arc/2` constraints while `M1` reduces by one the number of `seq/2` constraints but introduces a new `arc/2` constraint. Since the latter rule cannot be directly transformed in hardware with the proposed method, a transformation that operates on the constraints is needed. Indeed, since in rule `M1` the total number of constraints is left unchanged, there is no constraints propagation and hence the hardware implementation is still possible if we adapt the two type of constraints to a single one. Such constraints flattening is always possible using a new constraint with an arity equal to the greatest arity of the present constraints plus one. The added argument is intended for carrying a constant index associated with the former constraint type. In the bottom part of Figure 4.9 the constraint `c/3` is used for replacing `seq/2` and `arc/2`, and the constants `0` and `1` are adopted as the respective indexes.

Both RHBs associated with the two rules have two inputs and two outputs. The first receives two signals, corresponding to two `c/3` constraints, checks the guard (including the constrain type equal to `0`) and, eventually, rewrites one constraint. The second takes two `c/3` constraints as well and, if the guard holds, rewrites both of them.

The PHB simply consists of a hardware block with two inputs and two outputs as well. As in the case of the running example (see in particular Example 4.2.2)

Figure 4.9: Merge sort algorithm

```
M0    @  arc(X,A) \ arc(X,B) <=> A<B | arc(A,B).
M1    @  seq(N,A), seq(N,B) <=> A<B | seq(N+1,A), arc(A,B).
```

$$\Downarrow$$

```
M0    @  c(0,X,A) \ c(0,X,B) <=> A<B | c(0,A,B).
M1    @  c(1,N,A), c(1,N,B) <=> A<B | c(1,N+1,A), c(0,A,B).
```

both rules need to be duplicated since they have two constraints of the same type in their head. Hence the PHB includes four RHBs whose outputs are committed according to the validity of their guards. The results of the rules are committed following the textual order since after the constraint flattening no parallelization is possible between them. In Section 5.3 we will discuss in detail all the possible degree of parallelization inside a PHB.

The actual parallelization is performed by the CS that pairs the query constraints and assigns each couple to a different instance of the PHB. The CS waits the end of the computation of all the PHBs before starting to scramble the couples of constraint and to reassign them to the PHBs. Unlike the running example, there is no so much difference among the computational times of each PHB because each rule can be applied only once to the same set of constraints in a PHB. In other words the feedback mechanism from the outputs to the inputs of a RHB is not exploited. Indeed the constraints produced by each rule cannot satisfy the guard, when coupled with one of the constraint that had generated it, even if it matches the head of the rule. For instance in rule `M1`, the second argument of the body constraint `c(1,N+1,A)` is, by definition, different from the one of the head constraints `c(1,N,A)` and `c(1,N,B)`. Hence the difference in the computational time is only due to the fact that the rule can fire or not. In Section 5.4 we will show how to further parallelize the rules execution adopting two separate frameworks for `M0` and `M1`. Furthermore the experimental results of such implementations will be showed.

5 Improving Parallelization

In this Chapter we investigate alternative hardware architectures, that address the possibility of increasing the degree of parallelization present in the generated hardware blocks, thanks to the CHR properties reviewed in Section 2.2.3.

Besides the general framework described in Chapter 4 we want to propose optimized frameworks in order to speed up the computation in presence of algorithms that considerably reduce the number of constraints during computation. In such cases it is worth noting that a CS, that simply combines all the constraints in all the possible combinations, is highly inefficient. In fact many constraints that have been removed by the PHBs still continue to be shuffled by the CS uselessly.

With the aim of facing the problem of time efficiency two optimizations are proposed (in Section 5.1 and 5.2) relying on different degrees of parallelization. In particular we will see how with simple changes in the hardware framework the property of strong parallelism can be exploited. Furthermore the adoption of a set based semantics for CHR can open the door to an even more wide parallelism. In Section 5.3 we will discuss the rule parallelism at the PHB level pointing out the possible rules dependency that can lead to a concurrent execution. Finally, in Section 5.4, the online property of CHR is used for boosting the computation of a merge sort algorithm. Experimental results of practical implementations will be given along the Chapter.

5.1 Strong parallelism

The proposed approach to solve the issue derived from the fixed nature of CS relies on the possibility of exploiting the strong parallelism property of CHR (introduced in Section 2.2.2), that assumes that rules can work on common constraints at the same time if they do not rewrite them.

We need a new hardware block charged to combine, in parallel on several PHBs, the kept constraints with different sets of removed constraints. An example of such device can be provided optimizing the CS obtained by the implementation of the gcd rules. Figure 5.1 shows a possible implementation for a five constraints query but the design can be easily increased linearly with the number of constraints. It relies on a circular shift register preloaded with the query constraints and with one

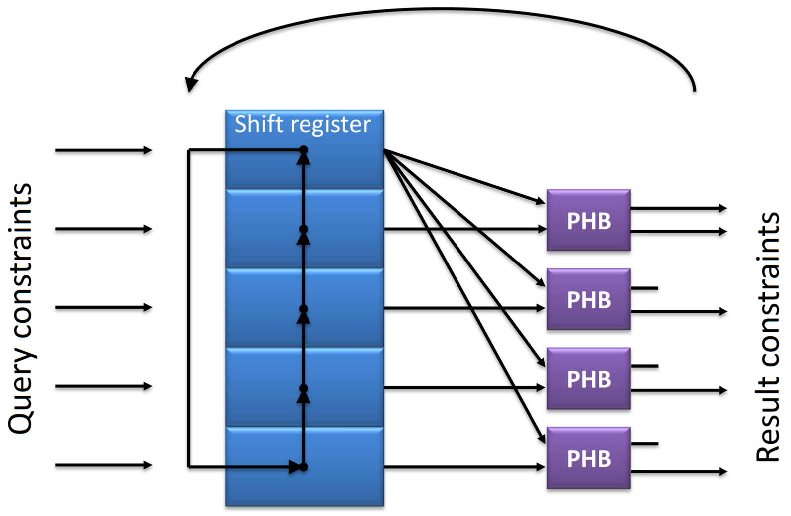


Figure 5.1: Hardware design scheme for strong parallelism

cell connected to all the first input (kept constraint) of the PHBs and all the others connected to the second input (removed constraint) of each PHBs. Each time the PHBs terminate their computation the new output constraints replace the old ones in the shift register and they shift until a *valid* constraint fills the first position of the register (we skip the steps with a not *valid* constraint in the first position). Using this topology there is no need to implement multiple instances of the same rule at the PHB level (see Section 4.2.2): indeed now the order of the inputs constraints matters because one is the kept `gcd` and the other is the removed one. As consequence, apart from the first PHB, the output carrying the kept constraint can be left disconnected because it refers always to the same constraint.

An experimental result of the proposed strong parallel architecture was already presented in Section 4.4. The reduction in execution time is relevant over all the measurements we carried out with different number of constraints, reaching up to one order of magnitude of speed up. The area needed in the FPGA for such optimization is basically equivalent to the one needed by the general architecture. Despite an increased number of PHBs ($n - 1$ instead of $n/2$, where n is the number of query constraints) the occupied area in FPGA to implement such optimization remains more or less constant. This is due to the more optimized structure of the shift register with respect to the CS. Indeed the CS is implemented as a finite state machine with a very large number of states S calculated in Eq. 4.3. Clearly a more optimized CS (for instance with a look up table where all the permutations are stored) would lead to an area saving, but the dependency of its dimension to S will remain unchanged.

Example 5.1.1. As we did in Example 4.2.3 for the CS, here we show an instance of computation of the gcd algorithm employing the shift register architecture and starting from the same sample query:

Table 5.1: Gcd derivation in hardware with strong parallelism

	PHB_1	PHB_2	PHB_3	PHB_4	PHB_5
$SR\{(6,v),(12,v),(45,v),$ $(15,v),(9,v),(3,v)\}$ R1 R1 R1 R1 R1 R0//R1 R1 R1 ∅ R1 ∅ R1 ∅ ∅ R1 ∅ R1 ∅ ∅ R1 ∅ R1 ∅ ∅ R1 ∅ R1 ∅ ∅ ∅ ∅ R1 ∅ ∅ ∅	$\{gcd(6),v\},\{gcd(12),v\}$ $\{gcd(6),v\},\{gcd(6),v\}$ $\{gcd(6),v\},\{gcd(0),n\}$	$\{gcd(6),v\},\{gcd(45),v\}$ $\{gcd(6),v\},\{gcd(39),v\}$ $\{gcd(6),v\},\{gcd(33),v\}$ $\{gcd(6),v\},\{gcd(27),v\}$ $\{gcd(6),v\},\{gcd(21),v\}$ $\{gcd(6),v\},\{gcd(15),v\}$ $\{gcd(6),v\},\{gcd(9),v\}$ $\{gcd(6),v\},\{gcd(3),v\}$	$\{(gcd(6),v\},\{gcd(15),v\}$ $\{gcd(6),v\},\{gcd(9),v\}$ $\{gcd(6),v\},\{gcd(3),v\}$	$\{gcd(6),v\},\{gcd(9),v\}$ $\{gcd(6),v\},\{gcd(3),v\}$	$\{(gcd(6),v\},\{gcd(33),v\}$ $\{(gcd(6),v\},\{gcd(27),v\}$ $\{(gcd(6),v\},\{gcd(21),v\}$ $\{(gcd(6),v\},\{gcd(15),v\}$ $\{(gcd(6),v\},\{gcd(9),v\}$ $\{(gcd(6),v\},\{gcd(3),v\}$
$SR\{(0,n),(3,v),(3,v),$ $(3,v),(3,v),(6,v)\}$					
$SR\{(3,v),(3,v),(3,v),$ $(3,v),(6,v),(0,n)\}$ R1//R0 R1//R0 R1//R0 R1 ∅ ∅ ∅ ∅ R1//R0 ∅	$\{gcd(3),v\},\{gcd(3),v\}$ $\{gcd(3),v\},\{gcd(0),n\}$	$\{gcd(3),v\},\{gcd(3),v\}$ $\{gcd(3),v\},\{gcd(0),n\}$	$\{gcd(3),v\},\{gcd(3),v\}$ $\{gcd(3),v\},\{gcd(0),n\}$	$\{gcd(3),v\},\{gcd(6),v\}$ $\{gcd(3),v\},\{gcd(3),v\}$ $\{gcd(3),v\},\{gcd(0),n\}$	$\{gcd(3),v\},\{gcd(0),n\}$

```
    gcd(6), gcd(12), gcd(45), gcd(15), gcd(9), gcd(33).
```

In such case we need five PHBs working on a common constraint placed in the first cell of a six cells shift register. We indicate as $SR(a, b, c, d, e, f)$ the current status of the shift register: each cell, denoted with a letter $a \dots f$, is occupied by one pair (constraint, valid signal).

In Table 5.1 the full computation is reported: all the derivation is carried out in just two steps of the shift register (one is skipped because the constraint in the first cell is marked as not valid). When the second input of the PHBs becomes smaller than the first input the partial computation halts (even if both constraints are valid) because the first input cannot be modified since it represents the kept constraint.

It is worth noting that, to fully exploit strong parallelism, more complex hardware blocks could be required, but it is always implementable in an hardware solution (in the worst case using look up tables to perform constraints pairing like in the case of the CS). For instance, the CHR rule implementing the Floyd-Warshall algorithm 4.6 has three constraints in its head thus the design in Figure 5.1 can be directly adopted, but it does not fully employ strong parallelism. Since two out of three head constraints are kept, strong parallelism gives the possibility of applying concurrently $n - 2$ rules where n is the number of query constraints. A straightforward application of the simple shift register for this rule allows for a parallelization of just $(n - 1)/2$ rules. A circular shift register used in the configuration of Figure 5.2 can pairs the two kept constraints in order to gain the greatest degree of parallelization. There are as many cells as the query constraints and one cell is kept hold while the others shift following the arrows direction. At every step of the shift register $n/2$ couples (denoted in figure by blue rectangles) are created. Every couple of constraints can feed $n-2$ PHBs because the third input can be chosen among the remaining constraints. Since the shift register has $n - 1$ cells the total number of possible tern of input constraints for a PHB is then $(n/2)(n-2)(n-1) = \binom{n}{2}(n-2)$, the number of combination of n distinct query constraints matching the two kept constraints combined with the remaining removed constraint.

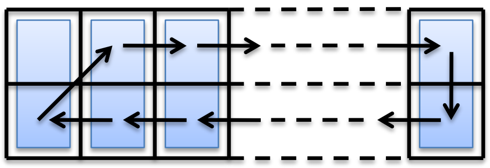


Figure 5.2: Circular shift register for pairing two constraints

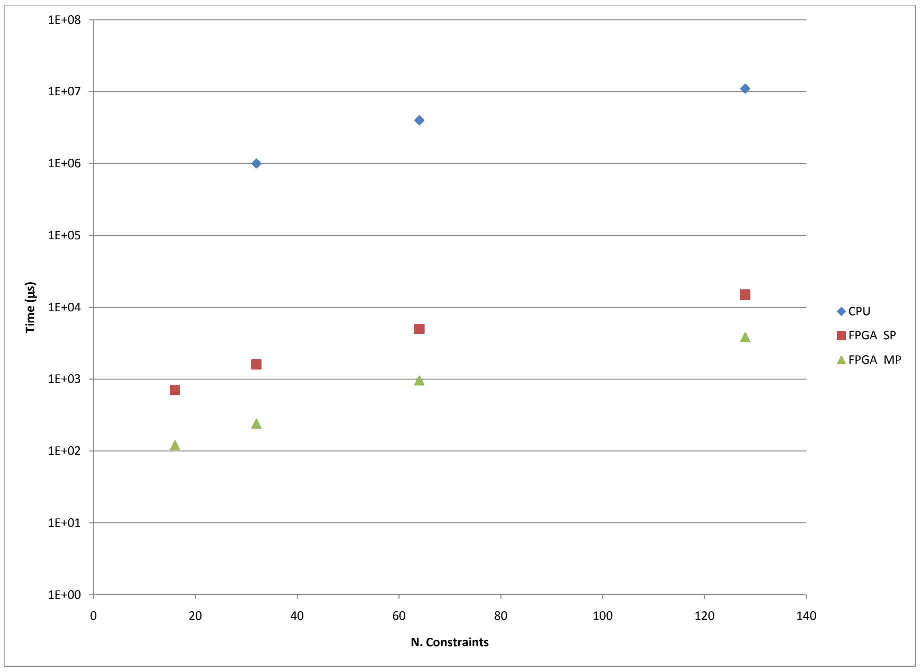


Figure 5.3: Prime execution time (log scale)

5.2 Massive parallelism

The set-based semantics CHR^{mp} formally expressed in [84] is based on the idea that the constraints can be considered as multiplicity-independent objects and we can have additional copies of them at our disposal. In such a context a duplicate removal of one constraint can be replaced by the removal of two copies of the same constraint. The degree of parallelism introduced by that change of perspective is extremely high because it completely gets rid of the conflicts generated by the removal of the same constraint by multiple copies of one rule. Moreover it preserves a constraint removed by a rule that can be necessary for the application of another instance of the rule. The main drawback of CHR^{mp} relies on the fact that it loses the soundness with respect to the sequential execution when the program is not deletion-acyclic that means that two distinct constraints cannot be responsible for their mutual removal.

CHR^{mp} can apply very well to the algorithms that considerably reduce the number of constraints. Indeed a typical class of programs that can benefit from set-based semantics is shown to be the filter algorithms. We can consider as an example the simple program that extracts the prime numbers from a set of integers:

```
Prime @   prime(X) \ prime(Y) <=> Y mod X = 0 | true.
```
(5.1)

Prime rule belongs to the CHR subset defined by eq. 4.1 as the number of `prime` constraints decreases every time the rule fires and there is no introduction of new

Table 5.2: Derivation of the prime program in hardware with massive parallelism: PHBs computation

PHB_1	PHB_2	PHB_3
$\{prime(7), v\}, \{prime(3), v\}$	$\{prime(7), v\}, \{prime(21), v\}$	$\{prime(7), v\}, \{prime(15), v\}$
	$\{prime(7), v\}, \{prime(21), n\}$	

PHB_4	PHB_5	PHB_6
$\{prime(3), v\}, \{prime(21), v\}$	$\{prime(3), v\}, \{prime(15), v\}$	$\{prime(21), v\}, \{prime(15), v\}$
$\{prime(3), v\}, \{prime(21), n\}$	$\{prime(3), v\}, \{prime(15), n\}$	

types of constraints. The prime program is also deletion-acyclic since there is no couple of numbers where one is a multiple and a submultiple of the other one at the same time. Finally the property of strong parallelism can apply as well because multiple instances of the rule can work on the same `prime` constraint that is kept.

The proposed architecture in case of massive parallelization relies on a number $\binom{n}{k}$ of PHBs where n is the number of query constraints and k is the number of inputs of each PHB that computes all the possible combinations of input constraints just in one step. Then n AND gates with $n-1$ input collect the *valid* signals of all the instances of each constraint. With such connection scheme we can assure that only the never removed constraints will be kept.

Example 5.2.1. As an example we want to explicitly show the simple massive parallel execution of the prime program expressed by rule 5.1. Given the test query:

```
prime(7), prime(3), prime(21), prime(15).
```

the number of PHBs needed is $\binom{4}{2} = 6$ since the query constraints are four and the inputs for each PHB are two. In Table 5.2 is reported the parallel execution of each PHBs. PHB_2, PHB_4 and PHB_5 mark one of two input constrains as not valid while the other three PHBs leave the constraints unchanged. Table 5.3 reports the result of the four 3-input AND gates that collect the valid signal related to all the instance of each constraint. Only `prime(7)` and `prime(3)` are never marked as not valid, thus they are the only valid constraints at the end of the computation.

Table 5.3: AND table results for the derivation of prime program

	valid from PHBs			AND
prime(7)	v	v	v	v
prime(3)	v	v	v	v
prime(21)	n	n	v	n
prime(15)	v	n	v	n

5.2.1 Experimental results

As case study we generate all the hardware blocks needed by the proposed optimization for the prime program 5.1. We should notice that the built-in constraint used in the guard is not a built-in operator of standard VHDL. Indeed modulo like division can have as second operand only integers that are power of two since it corresponds in hardware just to a shift of one or more bits. Hence the modulo built-in has to be replaced by a hardware block that computes the division and returns as outcome the remainder. We used as part of the generated rule hardware block the Xilinx divisor core that takes ten clock cycles to output the remainder.

In Figure 5.3 the execution time of the implementation with the shift register (strong parallelism optimization) is tagged as FPGA SP while the one with the architecture mentioned above is the FPGA MP one. The improvement obtained corresponds about to an order on magnitude for queries with a low number of constraints and decreases with the rising of the number of constraints. This trend is due to the fast reaching of the physical bound of the hardware: indeed if we request a complete parallelization the occupied area of the FPGA will increase as $\binom{n}{k}$. Clearly even since $k = 2$ we obtain a quadratic growing that is not sustainable and a partial serialization is needed.

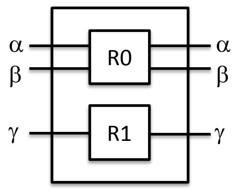
5.3 PHB parallelism

Until now we have seen how to apply weak parallelism 2.2(a) on the whole query through the CS, strong parallelism 2.3(a) through a circular shift register or massive parallelism thanks to a set semantics. In this Section we want to discuss the possibility of introducing parallelism at the PHB level. As we said in 4.2.2, the PHB is responsible of the parallelization of the CHR rules and, committing its outputs, it can perform different degrees of parallelization.

Example 5.3.1. Two RHBs can be weak or strong parallelized by its PHB depending on the constraints they modify. For instance in Figure 5.4 we present two programs consisting of a couple of rules that can be translated into the two corresponding PHB structures. For brevity the functions that bound the argument

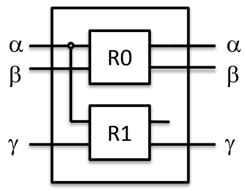
Figure 5.4: Examples of program that can benefit of weak (a) or strong (b) parallelism at the PHB level

```
R0 @ alpha(X) \ beta(Y) <=> beta(X).
R1 @ gamma(X) <=> gamma(Z).
```



(a)

```
R0 @ alpha(X) \ beta(Y) <=> beta(Z).
R1 @ alpha(X) \ gamma(Y) <=> gamma(W).
```



(b)

variables of the constraints and the guards are omitted. The corresponding PHB of the first program (a) relies on two separate RHBs that do not share any constraint, while the second program (b) corresponds to two RHBs that share the input associated with constraint `alpha`. In the second case the output `alpha` of one of the two RHBs can be left floating since both rules do not change its value.

Besides the two aforementioned parallelisms the PHB can implement another kind of rule parallelization. Since it tries to execute the rules in a predefined order (if no parallelization is possible) we can extend the domain of application of the strong parallelism. By definition of strong parallelism two rules can run in parallel if their heads overlap only in the kept part, but if we consider that the rules are applied in a particular order (for instance textual order) we can parallelize also two rules in which the kept part of the first is modified by the other. Adopting the same notation used for the weak and strong parallelism we define in Figure 5.5 the notion of priority parallelism. If the second rule modifies only the constraints kept unchanged by the first rule we can apply the two rules concurrently (in a single step). The proof is evident since it descends directly by the sequential application of the two rules. The second rule can act on the result of the first rule ($B \wedge E$). Clearly such parallelization cannot be done without assigning an execution order to the rules, indeed the application of the second rules would inhibit the first one leading to a different final state and to a not fair rules execution. The resulted layout of the PHB will have shared input constraints for the RHBs like the one in Figure 5.4(b), but, if the order of the rules does not allow priority parallelism, the output committing will be slightly different because it has to ensure the application of just one rule at a time.

Example 5.3.2. As an example of parallelism allowed by the ordered rule, we generate the PHB corresponding to the program consisting in the two rules in Figure 5.6(a). The two rules can be executed concurrently since they correspond to the

Figure 5.5: Priority parallelism in case of ordered rules

$$\frac{\begin{array}{c} A \wedge E \longmapsto B \wedge E \\ E \longmapsto F \end{array}}{A \wedge E \mathrel{|\!\!\!\Rrightarrow} B \wedge F}$$

priority parallelism in combination with monotonicity (Equation 2.18) expressed in Figure 5.6(b). The reason why the two rules can be easily executed concurrently in hardware relies on the fact that they are translated into two RHBs that drive just one signal (`beta` in rule `R0` and `alpha` in rule `R1`), thus no conflicts are generated on the outputs. Hence in this case the corresponding PHB commits the output of rule `R1` for `alpha` and `gamma`, and the output of rule `R0` only for `beta`.

Now let us suppose to want to generate the PHB for the program expressed by the same rules but in reverse order (`R0` after `R1`). In such case if rule `R1` fires the execution of rule `R0` in the same clock cycle is prevented (through appropriate *flags* in VHDL) and its application will be allowed again only in the next clock cycle if `R1` does not fire again.

Figure 5.6: Exemple of priority parallelism

```
R0 @ alpha(X) \ beta(Y) <=> beta(Z).
R1 @ gamma(X) \ alpha(Y) <=> alpha(W).
```

(a)

$$\frac{\begin{array}{c} A \wedge B \longmapsto A \wedge D \\ C \wedge A \longmapsto C \wedge E \end{array}}{A \wedge B \wedge C \mathrel{|\!\!\!\Rrightarrow} C \wedge D \wedge E}$$

(b)

5.4 Optimizing Merge sort

In Section 4.5 we have shown a straightforward implementation of the merge sort program that exploits only weak parallelism, in this Section alternative parallel architectures are presented. The merging operation of rule `M0` is a sequential operation, but the production of the `arc/2` constraints can be carried out in parallel to their merging. Trying to perform such parallelization at the PHB level would lead to a failure since the constraints flattening (needed by the transformation that associates a constraint to a hardware signal) would get a reduction of the number of rules that can fires concurrently since the increased number of checks in their guards. The matching between the constraints and the rule heads is treated as a strict and safe bound during the hardware compilation. If we remove it in order to increase the parallelism we produce as result just an increase of rules that cannot fire.

The proposed approach for parallelizing merge sort is, instead, based on the online property of CHR (see 2.2.2). Indeed the program can be naturally split into

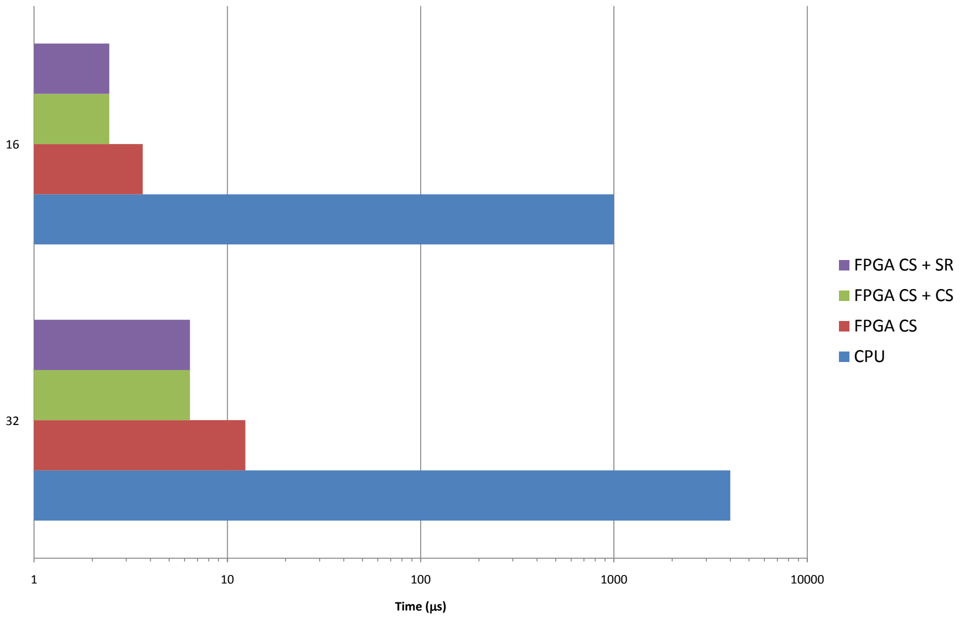


Figure 5.7: Merge sort execution time (log scale) for a query of 16 and 32 constraints

two parts corresponding to the two rules and some of the resulting constraints of one part can be used to populate runtime the constraint store of the second one. The `arc/2` constraints produced by rule `M1` are consumed only by rule `M0` while `seq/2` constraints are produced and consumed only by rule `M1`. If we consider the two rules as separate programs joined by the `arc` production and consumption, we can design a hardware constituted by two executors linked by a one-way buffer. Each rule corresponds to one full executor consisting of a CS and $n/2$ instances of a PHB containing the duplicated RHB, where n is the number of query constraints. The `seq/2` query constraints are loaded in the CS of the first executor and as new `arc/2` constraints are created (actually `c/3` constraint with the first argument equal to `0`), they are inserted in a buffer that feeds the CS of the second executor. Such CS at the beginning of the computation should be empty, hence it is preloaded with all the constraints marked as not valid and when it receives a new constraint from the buffer it replaces one of them. With this double architecture the two rules can rewrite constraints independently and asynchronously since the two executors are decoupled by the buffer.

5.4.1 Experimental results

With the same experimental setup adopted in Section 4.4 we implemented the merge sort algorithm with three different hardware designs. In Figure 5.7 a comparison between the execution times of such architectures is shown above the CPU execution reported as reference. With the label FPGA CS we refer to the execution time of a single executor of the complete program. This architecture corresponds to the one described in Section 4.5 that adopts only week parallelism. The execution times of the double executors with two CS are, instead, labeled FPGA CS + CS. In this configuration a FIFO is used to connect one cell of the first CS to one cell of the second. The two CS and the FIFO should have the same dimension since, in a normal execution, all the `seq` constraints (except the last that always remains unpaired) are converted in `arc` constraints. Instead the FIFO depth as to take into account the possibility that the second CS is not able to receive immediately the sent constraint because the receiving cell is occupied by a valid constraint. The last architecture tested (tagged FPGA CS + SR) replaces the second CS with a shift register like the one used in presence of strong parallelism (see Section 5.1). Indeed multiple instance of rule `M0` can be strong parallelized because one constraint of the head is kept and hence it can overlap on multiple rules.

The histogram clearly shows the speed up introduced by the adoption of the online property that gives the possibility of dividing the problem into two parts running in parallel. What can appear strange is the very good time agreements between the results of the two designs that use a double structure. The reason for a leak of speed up in case of strong parallelism in the second executor has to be charged in the fact that when the last `arc/2` constraint is generated, by the first executor, the partial result of the second one approximates very well the final result (or it is already that). Thus when the last constraint is retrieved by the second executor it has to apply just one rule before the actual end of the execution, hence the chosen parallelism for sorting the constraints does not matter.

6 Hybrid Compilation

In Chapter 4 we have seen how to implement a subset of CHR in hardware with strict bounds on the constraints propagation. The nature of hardware circuits imposes such limitations since it relies on a static and fixed architecture. To overcome the problem of dynamic allocation an addressable memory and a flexible controller are certainly needed. Obviously this architecture can be custom developed in reconfigurable hardware, but it would be an extraordinary effort not sufficiently paid off by the obtained performances with respect to the adoption of a general purpose processor and a standard RAM. In Section 6.2 we want to show how to interface a CPU with custom hardware implemented in FPGA in order to have the possibility of using the full CHR language boosted by an hardware accelerator.

Indeed often in software applications a small fraction of the source code consumes the most of the CPU resources while the remaining part focuses on initialization and system execution control. Code profiling can help identify the functions that consume the majority of the processor time and hardware co-processing is well suited to address the heavy computational load caused by such sub-tasks of the code especially if they allow for concurrency. FPGA accelerators are blocks of hardware that can be integrated into a processor-based system in order to offload some of the most computationally intensive tasks.

The hardware generator described in Chapters 4 and 5 can be seen from a higher perspective as a tool able to synthesize in hardware just few rules, belonging to a complex CHR program, that are charged of the most of the computational tasks. In such a context the limitation of using in hardware the CHR subset highlighted by rule 4.1 is not a restrictive bound because the processor can fill the hardware deficiency. In other words the processor produces constraints and the co-processor consumes or rewrites them.

Section 6.1 briefly reviews the architecture of standard hardware accelerators while in the introduction of HLLs in Section 3.3 we have already presented many programming languages adopted for such systems. Before dealing with the software/hardware partitioning we want to clarify which are the limitation encountered using the CHR subset identified in Section 4.1. Since the CHR code expressed by such subset can be deployed in hardware after a proper translation, in the rest of the thesis, we will refer to it as the synthesizable CHR.

The main restriction that we will address in this Chapter is the absence of propagation at all. Indeed it was stated that a synthesizable rule cannot introduce new constraints (not present in its head). Even if this limitation can appear very restrictive in Appendix B we will analyze the class of describable algorithms following the method employed by Sneyers in [95] and we will show that, despite the preamble, the outcome will be quite positive. Indeed in such paper it was shown that several subclasses of CHR are Turing-complete, in particular CHR with only one kind of rule.

Clearly, if our targets are performances, we cannot always afford a program translation that with a rules overhead, and a consequent increase of complexity, transforms the former program to a synthesizable one. Hence, even if the hardware execution of any CHR program is always possible (after a proper transformation), in Sections 6.2 we want to propose a method to overcome the language restrictions imposed by the synthesizable subset.

6.1 Related hardware accelerators

Concerning the employment of specialized hardware to aid the main computation, two mainstreams can be identified in the last decade, depending on the nature of the processing element coupled to the CPU:

1. Co-processors with full or reduced instructions set (ASIP) can cooperate with a general-purpose core. Clear instances of these computational platforms are the graphical processing units (GPUs), that are widely used as highly parallel and programmable co-processors [82], or the synergistic processing elements (SPEs) of the Cell Broadband Engine that are specialized offload co-processors [64].

2. Hardware accelerators based on reconfigurable device like Field Programmable Gate Arrays (FPGAs). Thanks to the versatility of such hardware component, the accelerator can be designed to perform a specific task as required. The CPU can easily talks to the accelerator through data and control registers or shared memory since the FPGA can directly sit on the processor bus. Hardware accelerator based on FPGA are widely exploited on different application fields for example cryptography [16], data processing [80] or computer vision[9]. We refer the reader to Section 3.1 for an overview of FPGA's architecture.

Actually, even if the majority of the systems relies on one of the above systems, the distinction is not so strict. Indeed we should mention, for example, the cases in which the FPGA can interact with the main processing core not only through the programming model interface but it is independent of and tightly coupled with the instructions set of the CPU [32].

Hardware designers have used FPGAs for many years as single-chip accelerated application but recently, due to the increased devices density, FPGAs are being the focus of system designers attention as well. This change of perspective sparked the debate on what could be the best language for describing a mixed hardware/software system. The lack of a suitable standard gave rise to a plethora of high-level languages (HLLs) oriented to increase the productivity in the description of reconfigurable computing devices [40]. Examples of platforms that adopt a HLL as input are the commercial Impulse-C [62] or the academic SPARK [53] or the more recent Lime [10] (a review of many of these languages can be found also in Section 3.3 since thy can be used as high-level hardware syntheses languages). With automated or semi-automated compilers and optimization tools the programmer is now assisted through a quick hardware accelerator prototyping and implementation. In such a way the need of a specific knowledge of hardware design methodology fades leaving the system programmer the possibility to focus on the entire project at a glance.

6.2 A hardware accelerator for CHR

In Chapter 4 it was shown how to synthesize hardware starting from a subset of CHR that does not take into account constraints propagation. Since dynamic allocation is not allowed in hardware due to physical bounds, such subset restriction may appear expected for a hardware designer but it sounds very restrictive for software programmer. In order to overcome the limitation on constraints propagation we studied a mixed (hardware/software) system where some rules, charged of the most computational task, are executed by specialized hardware (synthesized through the afore mentioned technique) while the remaining ones are executed by the main processor that can fill the hardware deficiency. The processor can easily take care of constraints production while the custom hardware can efficiently consume or rewrite them.

The hardware/software partitioning task is left to the programmer who should specify which rules have to be deployed in the hardware accelerator. A wrapper function virtually encapsulates those rules and actually provides for the data movement interface. It tailors the data to the hardware requirements and it enables their retrieval by the main program. The wrapper is then used as a call: when it is invoked some constraints are passed as arguments, then they are converted to a query for the hardware rules. The resulting constraints of the hardware execution are given back to the wrapper that introduces them in the main computation as return arguments. The wrapper can be seen by the programmer as a function for embedding low level instructions that speed up the program execution, even if it is a more complex service since it is not consisting of low level code but it is actually a call to a hardware driver. In Section 6.2.1 the details of the wrapper implementation are given.

In literature two kinds of modularity for CHR were studied and adopted: flat composition [5] and hierarchical composition. The former consists in a programs union with a simple merging of all the rules, while the latter allows for the reuse of

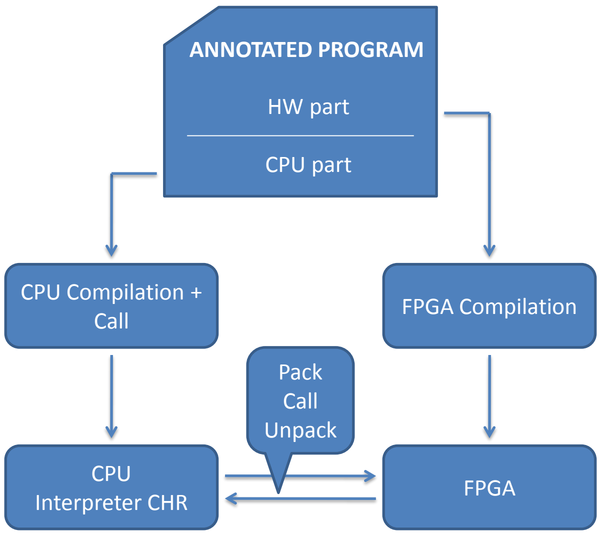


Figure 6.1: Hardware/software compilation diagram flow

CHR constraints turning them in built-in constraints for an other program. Hierarchical composition was applied in extending arbitrary solver [37] or CHR solver [90] with other CHR solver. An example of hierarchical composition similar to the one adopted by us is presented in [97], where an operation of minimum extraction is embedded in guards as CHR constraint. We use custom built-in constraints as wrappers in order to hold the execution of the main program, to run an other subprogram, and finally to restart the stopped execution.

After the presentation, in Section 6.2.1, of the double compilation path needed for compiling the software execution and the hardware synthesis, a detail description of the implementation of the CHR hardware accelerator is given through a case study in Section 6.2.2. Section 6.2.3 shows a worst case application of hardware/software partitioning and finally Section 6.2.4 gives a positive example of hardware accelerator applied to a typical problem of constraint solving.

6.2.1 Hardware/software compilation

The entire system compilation is split into two branches (see Figure 6.1) related to the software and hardware parts. We start the compilation from an annotated program in which the programmer highlights the rules that have to be executed by the hardware accelerator. The hardware compilation consists of the application of the method proposed in Section 4.2 that results in a bit stream directly deployable in a FPGA. On the other hand the standard software compilation will be necessary altered due to the removal of some rules from the full program specification. Since our implementation relies on a CHR system that adopts Prolog as *host language*, the execution of the removed rules will be embedded in a custom made built-in foreign Prolog predicate (the wrapper). When it is called all the constraints of a specific type are sent to the hardware accelerator that, once it has terminated the computation, will send the resulting constraints back to the constraint store. In the following Section we will make explicit this rule substitution in a practical case study implementation.

A foreign Prolog predicate is a C function that has the same number of arguments as the former predicate represented. C functions can provide for analyzing the passed terms, converting them to basic C types as well as instantiating arguments using unification. In concrete terms a C-source file can be loaded via the foreign interface including the special file `SWI-prolog.h` that defines various data types, macros and functions that can be used to communicate with SWI-Prolog. The possible actions from such interface to the Prolog environment are analysis of Prolog terms, construction of new terms, their unification, returning control information to Prolog, registering foreign predicates with Prolog, calling Prolog from C, recording database interactions and performing global actions on Prolog like halt, break, abort, etc. Foreign modules may be linked to Prolog using static or dynamic linking. In the former case the extensions, a file defining `main()` which attaches the extensions calls Prolog and the SWI-Prolog kernel distributed as a C library, are linked together to form a new executable. In the latter, instead, the extensions are linked to a shared library (`.so` file) and loaded into the running Prolog process.

A software architecture that handles the coprocessor is then embedded in such foreign Prolog predicate. The application software queues up packets of data to be sent to the coprocessor and provides a queue of buffers to be filled with return data. The application software also provides call-back routines that will be called by the driver whenever a packet of data is delivered to or received from the coprocessor. Either interrupt or polling mechanisms can be used by the processor to check if it needs to take any action. In such a way, the processor and coprocessor can both be active simultaneously. An estimation of the overhead introduced by the whole process of data handling is given in the next Section.

6.2.2 GCD matrix calculation

As case study we choose the calculation of the greatest common divisor (gcd) on a set of integers, to this end we virtually build a bi-dimensional triangular upper matrix whose elements will contain the gcd of pairs of integers belonging to the set. The rules reported in the upper part of Figure 6.2 are the CHR implementation of the matrix computation. We use the Euclid's algorithm (as in Example 2.2.1), expressed by rules `GCD0` and `GCD1`, to calculate the gcd between two integers and the propagation rules `Matrix0` and `Matrix1` to build the elements pairs from the initial set of integers. The constraint `set/2` has as first argument the order number of the set and as second one the value of the element. The first two arguments of `gcd/3` are used to denote the position of the element in the gcd matrix and the third is the respective value. Rule `GCD0` states that the `gcd/3` constraints with the value equal to zero can be removed from the store, while `GCD1` states that if two constraints `gcd(X,Y,N)` and `gcd(X,Y,M)` are present the latter can be replaced with `gcd(X,Y,M-N)` if `M>=N`. We can apply our translation technique to this program because the number of constraints remains bound during computation (exactly as the case of Example 4.1.1). Rule `Matrix0` produces only the upper half of the gcd matrix (due to the guard `X<Y`) since it is symmetric while `Matrix1` generates the diagonal elements that are trivially equal to the corresponding set element since the gcd of two equal numbers is the number itself. These two rules cannot be implemented in hardware because they are propagation rules that generate new `gcd` constraints.

In the hardware accelerator we deploy the functionality of rules `GCD0` and `GCD1` with the hardware blocks technique reported in Section 4.2. The remaining program running on the main processor consists of the two rules `Matrix0` and `Matrix1` with the addition of the rules reported in the bottom part of Figure 6.2. Rule `Pack` is intended to append all the constraints of type `gcd/3` to a *list* that has to be delivered to the hardware accelerator. `Call` is used to trigger the invocation of the custom Prolog predicate `hw_gcd/2` that is the actual responsible of the data transfer to and from the hardware accelerator. The constraint `call/0` is at disposal of the programmer to make the rule fire at the preferred time. For example we can add at the end of the query `call` if we wish that the `gcd` constraints were processed after the complete production of all of them. Finally the rule `Unpack` returns the constraints that are sent back from the hardware accelerator. In this particular example the application of such rule is not necessary because the output of the gcd computation

Figure 6.2: Gcd matrix program

```
GCD0    @   gcd(_,_,0) <=> true.
GCD1    @   gcd(X,Y,N) \ gcd(X,Y,M) <=> M>=N | gcd(X,Y,M-N).
```

```
Matrix0 @   set(X,N), set(Y,M) ==> X<Y | gcd(X,Y,N), gcd(X,Y,M).
Matrix1 @   set(X,N) ==> gcd(X,X,N).

                          ⇓

Pack    @   gcd(X,Y,N), list_in(L)#passive <=> list_in([(X,Y,N)|L]).
Call    @   call, list_in(L1) <=> hw_gcd(L1,L2), list_out(L2).
Unpack  @   list_out([(X,Y,N)|L]) <=> list_out(L), gcd(X,Y,N).
```

```
Matrix0 @   set(X,N), set(Y,M) ==> X<Y | gcd(X,Y,N), gcd(X,Y,M).
Matrix1 @   set(X,N) ==> gcd(X,X,N).
```

Figure 6.3: Derivation of the gcd matrix program in a software/hardware execution

`Matrix1`	`gcd(1,1,6)`
`Pack`	`list_in([(1,1,6)])`
`Matrix0`	`gcd(1,2,6), gcd(1,2,12)`
`Pack`	`list_in([(1,2,12),(1,2,6),(1,1,6)])`
`Matrix1`	`gcd(2,2,12)`
`Pack`	`list_in([(2,2,12),(1,2,12),(1,2,6),(1,1,6)])`
`Matrix0`	`gcd(1,3,6), gcd(1,3,45)`
`Pack`	`list_in([(1,3,45),(1,3,6),(2,2,12),(1,2,12),(1,2,6),`
	`(1,1,6)])`
`Matrix0`	`gcd(2,3,12), gcd(2,3,45)`
`Pack`	`list_in([(2,3,45),(2,3,12),(1,3,45),(1,3,6),(2,2,12),`
	`(1,2,12),(1,2,6),(1,1,6)])`
`Matrix1`	`gcd(3,3,45)`
`Pack`	`list_in([(3,3,45),(2,3,45),(2,3,12),(1,3,45),(1,3,6),`
	`(2,2,12),(1,2,12),(1,2,6),(1,1,6)])`
`Call`	`list_out([(3,3,45),(2,3,3),(1,3,3),(2,2,12),(1,2,6),`
	`(1,1,6)])`
`Unpack` x6	`gcd(3,3,45), gcd(2,3,3), gcd(1,3,3), gcd(2,2,12),`
	`gcd(1,2,6), gcd(1,1,6)`

will be constituted of just one constraint, but it is reported for generality purpose.

Example 6.2.1. The derivation of an exemplifying query for the revised version of the gcd matrix program in the lower part of Figure 6.2 is reported in Figure 6.3. The two rules deployed in hardware are `GCD0` and `GCD1`. We omit the hardware part of the derivation since it is very similar to the one presented in Example 5.1.1, if a strong parallel architecture is used. The sample query is:

```
list_in([]), set(1,6), set(2,12), set(3,45), call.
```

For clarity we do not show all the state transitions: in the first column the applied rule is reported while in the second one the introduced constraints are present. For a proper understanding of the derivation we should notice that the processor executes rules according to the refined operational semantics 2.2.1. Placing the `call/0` constraint at the end of the query the hardware execution is deferred at the end of the generation of all the `gcd/3` constraints. An alternative execution could fires the `Call` rule every time a `Matrix1` is executed. Such execution could be easily done adding `call/0` and `list([])` at the end of the body constraints of `Matrix1`, but the time performance would be certainly worse because of the transfer overhead introduced at every call.

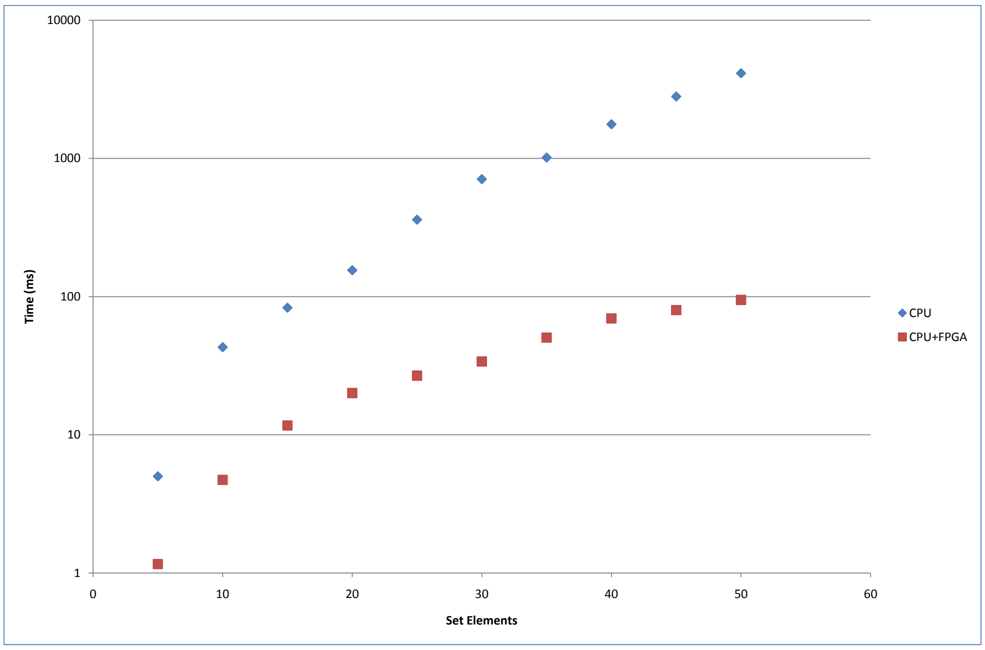


Figure 6.4: GCD matrix execution time (log scale)

Experimental Results

Co-processors and in general hardware accelerators typically need a data interface and a control interface: the first one moves data to and from memory while the second one is charged to set up the operations of the processing core. The overall system performance is mainly limited by the efficiency of the data movement, thus we paid attention to choose a proper hardware architecture of the data interface. Since the amount of transfered data required by our testbench is quite small, a single registers approach can easily fulfill our requirement. We exploit in the FPGA a PCI Express IP core (PCI-E IP) with the aim of making the hardware accelerator memory directly mapped in the main processor memory space. On the FPGA side it is profitable to include a memory buffer (FIFO) to act as a local cache to the accelerator core. In such a way, the processor is free to upload data to the FPGA buffer without worrying of being synchronized with the accelerator. The FIFO queue is implemented using an IP core that exploits the additional RAM blocks of the FPGA.

The hardware setup of the test bench relies on a Xilinx Virtex4 FPGA (xc4vfx60) running at 100MHz and connected to a PCI-E root complex of an ASUS P7P550 motherboard hosting an Intel Core i7 CPU running at 2.8GHz. On the software side we use the CHR system [89] for SWI-Prolog that lets us easily integrate memory mapping instructions thanks to the embedded interface to C [112] that we employ for the wrapper implementation.

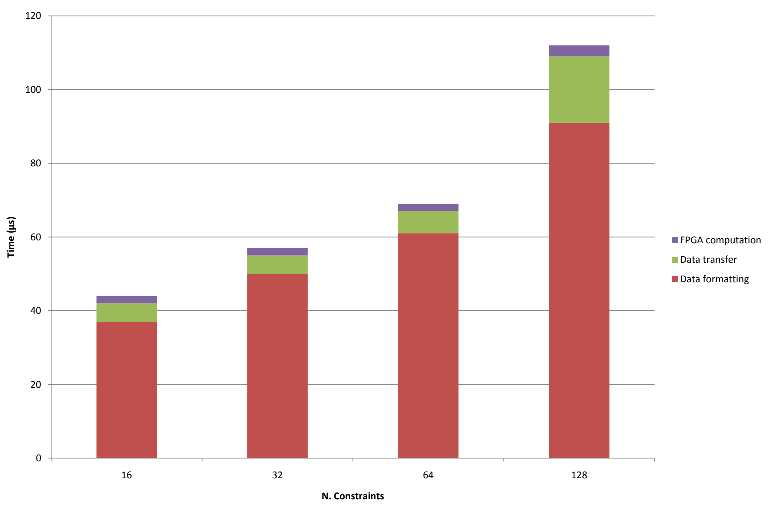


Figure 6.5: Additional time introduced by the hardware accelerator measured at different sizes of the query

We deploy in FPGA the firmware generated by the first two rules of Figure 6.2 bounding the number of constraints to 128 integers at most. The resulting hardware design relies on 127 PHBs deriving in parallel the gcd while a circular shift register pairs a constraint with all the others as described in Section 5.1.

In order to determine the total system execution time we used a single thread implementation in which the CPU is kept idle until the FPGA has performed its computation. Figure 6.4 compares the execution times of computation of the gcd matrix running both on the plain CPU and with the help of the FPGA (labeled CPU+FPGA in the plot). Even if the speed achieved is not comparable with the one obtained by the execution of the gcd algorithm entirely in FPGA (see Section 4.4), the execution time improvement is still in the range of one order of magnitude. This partial loss of gain is rewarded by a more flexible system thanks to the presence of the CPU.

A comparison of the extra time introduced with the addition of the hardware accelerator is presented in Figure 6.5 for 16, 32, 64 and 128 1-byte constraints. We measured the elapsed time as the sum of three different components: data formatting, data transfer and FPGA computation. The first one consists of the required time by a CHR rule for calling the foreign Prolog predicate that converts terms in basic C-type, arranges the constraint values in data packets, decodes the incoming packets and unifies the C-type values with terms. The remaining two components are respectively the routing time to send data through the PCI-E bus and the time needed by the FPGA for packing/unpacking and processing data. The measures show that the most expensive phase is mainly due to the data handling at the level of the CHR rule responsible of the built-in execution that sets up the FPGA

Figure 6.6: Preflow push algorithm

```
Push     @   h(U,UH), h(V,VH) \ e(U), res(U,V), phase(push) <=>
             UH>VH | res(U,V), e(V), phase(push).
Trans    @   phase(push) <=> phase(lift).
Lift1    @   e(U) \ h(U,HU), phase(lift) <=>
             U\=source, U\=sink, gemin(U), phase(update).
Mcalc    @   minimum(A) \ minimum(B) <=> A=<B | true.
Minit    @   getmin(U), res(U,V), h(V,H) ==> minimum(H).
Lift2    @   getmin(U), minimum(M), phase(update) <=>
             M1 is M+1, h(U,M1), phase(push).
```

computation. Clearly such burden of few microseconds per constraint is fully paid off by the speed up gained in the further concurrent execution of CHR rules in FPGA. The aforementioned results obtained implementing the Euclid's algorithm for the gcd calculation in hardware strengthen this thesis but with the following example we want to show what happens when the FPGA computation is less preeminent or even negligible wrt the CPU one.

6.2.3 Preflow push algorithm

Preflow push algorithm gives a solution of the maximum-flow problem: to find a feasible flow through a single-source, single-sink flow network that is maximum. The general algorithm works on a *flow network* defined as a direct graph $G = (V, E)$ of vertices V and edges E, where two vertices are called *source* and *sink* and at each edge a non-negative *capacity* c is associated. A *flow* is defined as a mapping $f : E \to \mathbb{R}^+$ that assigns to an edge a value that cannot exceed its capacity and obeys to the rule of conservation (the sum of the flows entering a vertex must equal the sum of the flows exiting a vertex, except for the *source* and the *sink*). A valid solution of the maximum-flow problem is given by a *flow* that maximizes the sum $\sum_{v \in V} f(v, sink)$ of the *flow network*.

We choose to implement the refined semantics version of the preflow push algorithm presented in [79]. The general algorithm is restricted to work on *flow networks* with capacities, between two vertices u and v, of the form $c(u, v) \in \{0, 1\}$ and $c(u, v) + c(v, u) \leq 1$. Positive excess of flow in a vertex is allowed during computation but it has to be zero (in order to make the flow compliant) by the termination. The flow excess is encoded as multiple copies of the constraint `e/1` that stands for one unit. Unit capacities make possible to represent each edge as a residual edge (constraint `res/2`) since if the flow is 1 the residual is 0 or vice versa. The constraint `h/2` associates with each vertex (first argument) a height (second argument). As reference, in Figure 6.6, the code of the algorithm is reported where we put in

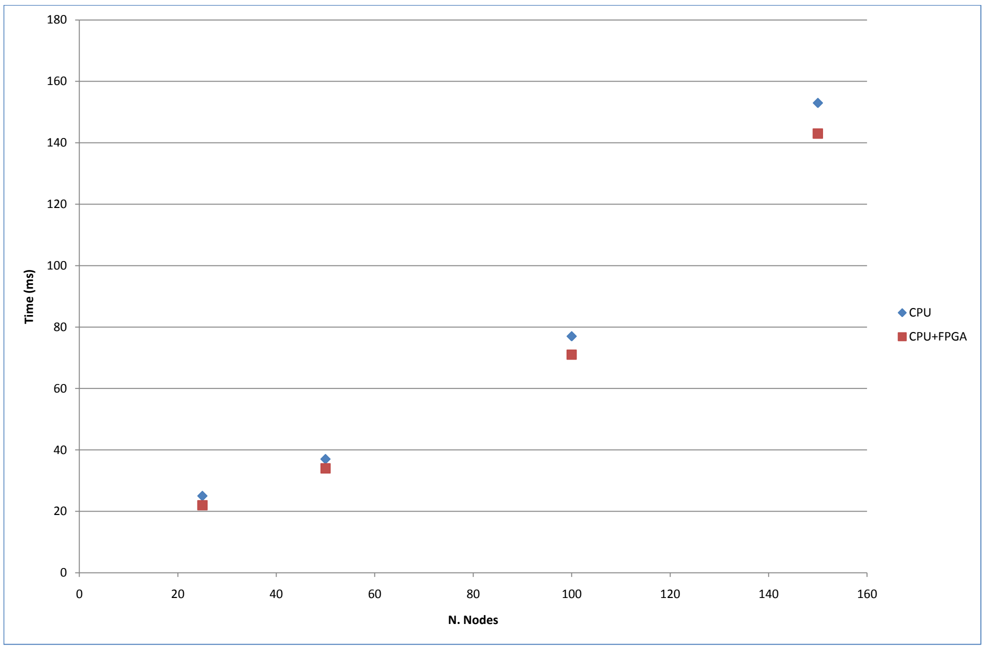


Figure 6.7: Preflow push algorithm execution time

evidence the rule that we deploy on FPGA. Constraint `phase/1` is used as control for executing one lift action when the `Push` rule is not applicable any more. In phase `update` the minimum height of the residual neighbors is calculated for each node by rules `Mcalc` and `Minit`. In particular `Minit` produces the `minimum\1` constraints that are filtered out by `Mcalc`.

Reducing rules, analyzed in Section 5.2, are efficiently implementable in FPGA because of the high degree of parallelization achievable. We are considering the deployment of rule `Mcalc` as a worst case for the advantages induced by the accelerator employment, because the rule maximizes the ratio between the number of sent constraints and the number of times multiple instances of the same rule are concurrently applied. If, for example, we use strong parallelism the ratio is always greater than 1 since for N constraints we need $M \leq N$ shifts of the register before getting the results. For the gcd example the ratio is less than 1 because the two rules are applied many times sequentially (at lest two) before the next shift of the register. This means that the denominator of the ratio is greater than N, except for the case in which all the `gcd` constraints have the same value.

The execution time of the preflow push algorithm is reported in Figure 6.7. The plot shows that, even in such worst case for the coupled system CPU and FPGA, the execution time is comparable with the one of a single CPU. This means that a huge number of function calls does not affect the performance of the program but rather they are well integrated in the compilation process of the software specification.

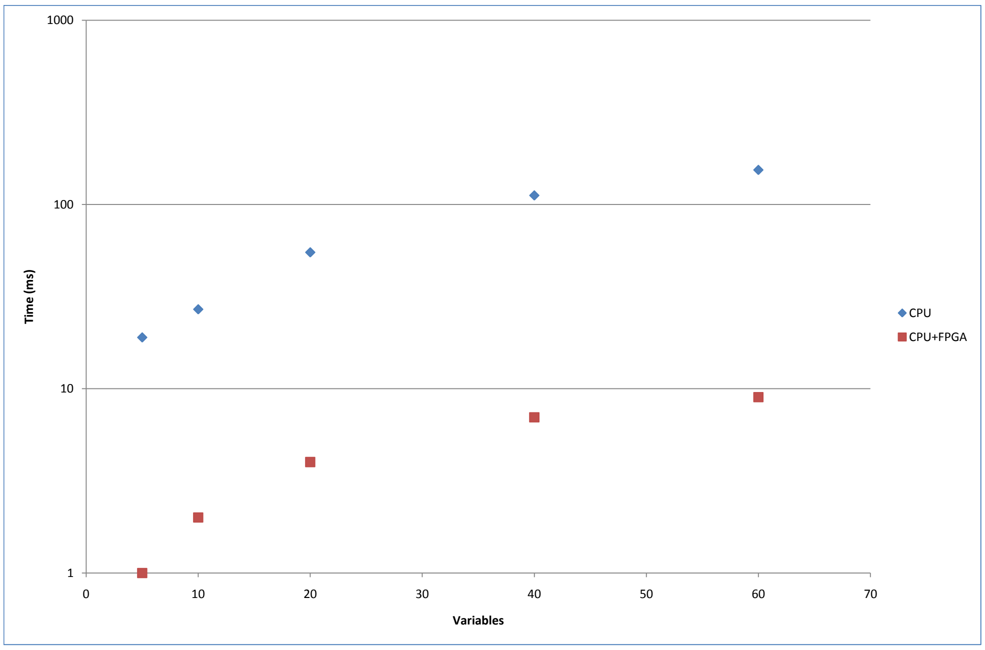


Figure 6.8: Interval domains solver execution time (log scale). The query was composed of 20 interval constraints and one arithmetic relation for each variable

6.2.4 Interval domains solver

Besides the case study presented in the previous paragraph we choose to implement a system that tackles a classical problem for constraint programming [12], a finite domains system. In CHR literature we can find several programs working on interval or enumeration constraints [49], we implemented a simple interval domains solver for bound consistency. As reference, in Figure 6.9, the code of the algorithm is reported where we put in evidence the first two rules that we deployed on FPGA. The solver uses the CHR constraint `::/2` for stating that a given variable can take

Figure 6.9: Interval domains solver algorithm

```
Redundant     @   X::A:B \ X::C:D <=> C=<A, B=<D | true.
Intersect     @   X::A:B, X::C:D <=> X::max(A,C):min(B,D).
-------------------------------------------------------------
Inconsistent  @   _::A:B <=> A>B | fail.
LessEqual     @   X le Y, X::A:_, Y::_:D ==> Y::A:D, X::A:D.
Equal         @   X eq Y, X::A:B, Y::C:D ==> Y::A:B, X::C:D.
NotEqual      @   X ne Y, X::A:A, Y::A:A <=> fail.
```

its value from a finite set notated with the custom operator `:`. For instance the constraint `X::a:b` means that the variable `X` can assume whatever integer value between `a` and `b`. `le/2`, `eq/2` and `ne/2` are CHR constraints as well and stand for the less or equal, the equal and the not equal operators, while `min` and `max` are Prolog built-in. Rule `Redundant` eliminates all the intervals that contain a subset interval of the same variable. `Intersect` replaces two intervals with their intersection by calculating the maximum of the lower bounds and the minimum of the upper bounds. `inconsistent` is used for point out whether an interval is not correctly defined (the lower bound is grater than the upper bound). Rules `LessEqual`, `Equal` and `NotEqual` represent the correspondent arithmetic order relations.

It can be easily noticed the complete absence of constraints propagation in the first two rules. Indeed `Redundant` removes one constraint without introducing any new one while `Intersect` introduces one constraint, but in return removes two of them. The only difference between these rules and all the previous implementations presented in this paper is the use of free logical variables instead of integers as arguments for the constraints. Clearly in hardware they cannot be treated as they stand, but, since during the whole program execution they are never bound, a substitution with indexes is possible during the step of packing and formatting the constraints to be sent to the accelerator. When the output constraints from the accelerator are received the indexes are replaced back with the corresponding logical variable.

The execution time of the interval domains solver is reported in Fig. 6.8. As in the case of the computation of the gcd matrix, the speed up obtained with the support of the hardware accelerator is over one order of magnitude along all the query range we had sampled.

It is worth noting that the hardware implementation of the two rules deployed in FPGA can employ only weak parallelism at the level of constraints assignment to the PHBs. Indeed the two rules cannot be strong parallelized since they do not share a kept part of the head. However the rules can benefit of the priority parallelism (see Section 5.3) at the PHB level. Indeed the two rules can be evaluated in a strict top to bottom order: given an interval, the removal of a larger interval that contains the former one, can always precede the evaluation of the intersection with a third interval. For instance from the three intervals (referring to the same variable) `3:6`, `2:8`, `5:7`, such parallelism lets evaluate the resulting interval `5:6` in just a single computational step.

Conclusions

We described the general outline of an efficient hardware implementation of a CHR subset able to comply with the restricted bounds that hardware imposes. The level of parallelization achieved provides a time efficiency comparable with that obtained with a design directly implemented in HDL. At the same time, the proposed solution offers a more general framework reusable for a wide range of tasks. Additionally, it was shown that, applying the same hardware generation technique to CHR with HDL built-in operators, we obtain elementary hardware blocks that can be easily integrated with existing HDL code. Indeed a smooth integration is guaranteed by the modularity of the HDL language (target of the source to source compilation) at a twofold level: in the behavioral and structural representation of the hardware.

Clearly this thesis does not claim to be an answer to the problem of the high level hardware synthesis, but it attempts to highlight how a declarative programming language like CHR can contribute to ease the burden of hardware designers. Several examples of simple CHR programs showed the generality of the proposed framework and how, with simple changes in a few rules, we are able to compile a specific specialized hardware able to execute the CHR code for solving completely different tasks. Reconfigurable hardware made it possible to easily obtain in short time the corresponding hardware circuits. The adoption of an FPGA as testing ground resulted to be a versatile solution for achieving a significant speed up of the CHR execution.

Different degrees of parallelization naturally embedded in CHR were pointed out and fully exploited thanks to the development of custom hardware structure. Even if the concurrency in CHR is well known, in the literature only few cases are present where it was actually turned into parallelism. The reason for this is, in our opinion, to be searched not only in the lack of parallel models, but in the lack of proper hardware architectures. The proposed optimizations to the general framework presented in this thesis deal also with properties that not necessarily involve concurrency like the *online* property, or with particular set semantics that revealed that for certain classes of algorithms the degree of parallelization achievable in hardware is only limited by the available space resource.

The proposed hardware compilation was validated with several examples that show how to produce the hardware needed to execute classical algorithms like merge sort, Euclid's algorithm, or an interval domains solver. In such case studies the evident expressive power of CHR has been combined with the time efficiency peculiar to simple low level hardware structures.

In several parts of the thesis it was shown how the fixed nature of the hardware is a critical point for the hardware translation of a CHR program. Propagation is not strictly needed for the completeness of the language, but it is clearly a very important feature for its complexity and expressive power. For this reason we proposed a classical CHR executor coupled with a hardware accelerator dedicated to simple tasks like the fast rewriting of some constraints. Such hardware based approach can increase the performance of constraint programming trying to achieve a stronger coupling between algorithms and platforms. A custom hardware based on FPGA was used as accelerator for CHR code: the traditional program execution was then interfaced with a custom accelerator engine.

We presented case studies in which the high level hardware compilation produced the needed hardware code for the accelerator, and simple CHR rules, added to the program, allowing the CPU to cooperate with the accelerator itself. The trade off raised by the data transfer overhead (between processor and hardware accelerator) and the speed up achieved in hardware was measured and later analyzed by means of a worst case example of hardware/software partitioning. Finally we should remark that the system integration was made smoother by the adoption of the same language for the software and hardware description suggesting how CHR could be adopted as a system level language.

Further work

Further improvements to the general framework, especially in terms of applicability also to problems where the number of constraints does not necessarily decrease during the computation, will be subject to future research. However an important challenge will be the hardware implementation of complex programs as well. A general treatment of rules dependency at the PHB level is still missing and only appropriate considerations on rules interaction can lead to a hardware performing parallel execution, pipelining and balancing out circular dependencies. We have seen that the hardware parallelization can be efficiently done at different levels of the design, the development of a theoretical model, able to clarify the most favorable strategy, would be useful for developing a general parallelization paradigm. These studies can eventually open the door to the production of a source-to-source compiler that takes as input CHR and carries out a structural hardware description in HDL ready for a synthesis tool.

Regarding the hardware accelerator for speeding up the CHR execution, we should mention the possibility of automating the process of rule selection for the hardware deployment. Results coming from a profiler could help a static analysis on the CHR program to identify the rules that are the most expensive to be executed. Moreover our preliminary tests carried out on simple CHR programs could be extended to more complex application.

On the hardware side this extension can lead to the adoption of a different approach in the data communication between the the processor and the accelerator. Indeed the slave register schema we adopted can be improved in a master attachment one thanks to a DMA engine on the FPGA that offloads the data directly to the main memory. Concurrent computations of the accelerator and the processor could then be explored. Further works will also investigate the possibility of fully exploiting the capability of reconfigurable hardware: the FPGA could then be totally or partially reprogrammed run-time with different sets of rules. If the reconfiguration costs will turn out to be sustainable for the entire program execution the issue of the space required by the rules parallelization will be partly resolved.

A
Program listings

This Appendix contains the program listings of some examples of implementations presented in Chapters 4 and 5.

List A.1 reports the VHDL description of the hardware blocks needed to form the PHB of the gcd program reported in Example 4.2.2. The first part of the code shows the *entity* declaration of `gcd` (see Example 3.2.4) that contains the input and output signals of the PHB. Besides the signals related to the RHBs `gcd`x, `valid`x, `gcd_out`x and `valid_out`x, the port listing contains the synchronization signals provided by the PHB: `clk`, `reset` and `finish`. The *architecture* of `gcd` has four *processes* executed in parallel, called `r0_1`, `r0_2`, `r1_1` and `r1_2`, that correspond to the four RHBs in Figure 4.4. In particular they correspond to the two instances required by Equation 4.2 of rule `R0` and `R1` presented in Figure 4.3. The committing part of the PHB is carried out by the *variable* `flag` that gives a priority to the PHB outputs assignment. Finally the *process* `finish_p` is charged to rise the finish signal when the output signals cannot be further modified by the other *processes*.

List A.1: PHB of the gcd program

```
entity gcd is
    Port ( clk : in  STD_LOGIC;
           reset : in STD_LOGIC;
           gcd1 : in  STD_LOGIC_VECTOR (7 downto 0);
           gcd2 : in  STD_LOGIC_VECTOR (7 downto 0);
           gcd_out1 : out  STD_LOGIC_VECTOR (7 downto 0) :=X"00";
           gcd_out2 : out  STD_LOGIC_VECTOR (7 downto 0) :=X"00";
           valid1 : out STD_LOGIC;
           valid2 : out STD_LOGIC;
           valid_out1 : out STD_LOGIC := '1';
           valid_out2 : out STD_LOGIC := '1';
           finish : out STD_LOGIC := '0');
end gcd;

architecture Behavioral of gcd is
  signal gcd1_sig : std_logic_vector (7 downto 0) := X"00";
  signal gcd2_sig : std_logic_vector (7 downto 0) := X"00";
  signal valid1_sig : std_logic := '1';
  signal valid2_sig : std_logic := '1';
  signal finish_sig : std_logic := '0';
  signal finish_sig_reg : std_logic := '0';
  signal gcd1_sig_reg : std_logic_vector (7 downto 0) := X"00";
  signal gcd2_sig_reg : std_logic_vector (7 downto 0) := X"00";
  shared variable flag : std_logic := '0';
  shared variable finish_flag : boolean := false;

begin

  r1_1: process (clk, reset, gcd1, gcd2, gcd1_sig, gcd2_sig, valid1_sig,
                 valid2_sig)
  begin  -- process r1_1
    if reset = '1' then
      gcd2_sig <= gcd2;
    elsif (clk'event and clk='1') then
      if (valid1_sig='1' and valid2_sig='1') then
        if gcd2_sig>=gcd1_sig then
          gcd2_sig <= gcd2_sig - gcd1_sig;
          flag := '1';
        else
          flag := '0';
        end if;
      end if;
    end if;
  end process r1_1;

  r1_2: process (clk, reset, gcd1, gcd2, gcd1_sig, gcd2_sig, valid1_sig,
                 valid2_sig)
  begin  -- process r1_2
    if reset='1' then
      gcd1_sig <= gcd1;
    elsif (clk'event and clk='1') then
      if (valid1_sig='1' and valid2_sig='1') then
        if flag='0' then
          if gcd1_sig>=gcd2_sig then
            gcd1_sig <= gcd1_sig - gcd2_sig;
          end if;
        end if;
      end if;
    end if;
  end process r1_2;

  r0_1: process (clk, reset, gcd1, gcd2, gcd1_sig, gcd2_sig, valid1_sig,
                 valid2_sig)
  begin  -- process r0_1
    if reset = '1' then
      valid1_sig <= valid1;
    elsif (clk'event and clk='1') then
      if gcd1_sig=X"00" then
      valid1_sig <= '0';
    else
      valid1_sig <= '1';
    end if;
  end if;
end process r0_1;

r0_2: process (clk, reset, gcd1, gcd2, gcd1_sig, gcd2_sig, valid1_sig,
              valid2_sig)
begin  -- process r0_2
  if reset='1' then
    valid2_sig <= valid2;
  elsif (clk'event and clk='1') then
    if gcd2_sig=X"00" then
      valid2_sig <= '0';
    else
      valid2_sig <= '1';
    end if;
  end if;
end process r0_2;

gcd_out1 <= gcd1_sig;
gcd_out2 <= gcd2_sig;
valid_out1 <= valid1_sig;
valid_out2 <= valid2_sig;

finish <= '1' when finish_sig='1' and finish_sig_reg='0' and
                   finish_flag else
          '0';

finish_p: process (clk, reset, gcd1_sig, gcd2_sig, finish_sig)
begin  -- process finish_p
  if reset='1' then
    gcd1_sig_reg <= X"00";
    gcd2_sig_reg <= X"00";
    finish_sig_reg <= '0';
    finish_flag := true;
    finish_sig <= '0';
  elsif (clk'event and clk='1') then
    gcd1_sig_reg <= gcd1_sig;
    gcd2_sig_reg <= gcd2_sig;
    finish_sig_reg <= finish_sig;
    if gcd1_sig=gcd1_sig_reg and gcd2_sig=gcd2_sig_reg then
      finish_sig <= '1';
    else
      finish_sig <= '0';
    end if;
  end if;
  if finish_sig='1' then
    finish_flag := false;
  end if;
  end process finish_p;

end Behavioral;
```

List A.2 reports the main *processes* constituting a possible implementation of the CS described in Section 4.2.3. For brevity the *entity* and the rest of the *architecture* are omitted. The two *processes* `FSM_FF` and `FSM_LOGIC` are used to describe the finite state machine that controls the signals switch. Basically it first waits for the `start` signals coming from all the PHBs and then it permutes the output signals enabling the execution of *process* `comb_p`. This latter *process* is the actual responsible of the switching between the input signals `const_sig` and the output ones `query_sig`. It employs the circular shift register scheme of Figure 5.2 for pairing 128 signals each other in order to result in 64 different couplets each step. The switching of the input/output signals coincides also to the rising of the `finish_sig` signal that triggers the rising of the reset signals for the PHBs.

List A.2: Processes of the CS for a query of 128 constraints

```
FSM_FF: process (clk, reset, next_state)
  begin
    if reset='1' then
      state <= init;
    elsif clk'event and clk='1' then
      state <= next_state;
    end if;
  end process FSM_FF ;

  comb_p: process (clk, reset)
  begin
    if reset='1' then
      const_sig <= query_sig;
    elsif clk'event and clk='1' then
      if finish_sig = '1' then
        const_sig(0) <= query_sig(0);
        const_sig(1) <= query_sig(3);
        const_sig(2) <= query_sig(1);
        const_sig(3) <= query_sig(5);
        for i in 1 to 61 loop
          const_sig(2+i*2) <= query_sig(2*i);
          const_sig(3+i*2) <= query_sig(5+2*i);
        end loop;  -- i
        const_sig(126) <= query_sig(124);
        const_sig(127) <= query_sig(126);
      else
        const_sig <= query_sig;
      end if;
    end if;
  end process comb_p ;

  FSM_LOGIC: process (clk, reset, state, start, query_sig, valid_sig)
  begin  -- process FSM_LOGIC
    next_state <= state;
    case state is
           when init => start_sig <= '1';
                        reg <= (others => '0');
                     next_state <= start;
           when start => finish_sig <= '0';
                           start_sig <= '0';
                           for i in 0 to 63 loop
                             if start(i)='1' then
                               reg(i) <= '1';
                             end if;
                           end loop;  -- i
                           if reg = (reg'Range => '1')  then
                             next_state <= mux;
                           end if;
           when mux =>
                        next_state <= stop;
           when stop => finish_sig <= '1';
                        reg <= (others => '0');
                        next_state <= start;
           when others => next_state <= start;
    end case ;
  end process FSM_LOGIC;
```

The code of List A.3 is the implementation of the architecture described in Section 5.1 for 128 signals. As for the previous List only the main *processes* are reported. `FMS_p` controls the finite state machine that handles the PHBs synchronization and enables the circular shift register that follows the scheme of Figure 5.1. Such shift register is implemented in *process* `shift_p` and is enabled by signal `shiftreg_sig` each time the finite state machine receives all the `start` signals from the PHBs of when it finds a not valid signal in the first cell (`valid_reg(0) = '0'`). Finally, when the signal in first cell is valid the reset signal is broadcast to all the PHBs with the rising of `finish_sig`.

List A.3: Processes of the shift register for a query of 128 constraints

```
shift_p: process (clk, reset, query_sig, valid_sig, shiftreg_sig, valid_reg)
  begin
    if reset='1' then
```

```
      shiftreg_sig <= query_sig;
      valid_reg <= valid_sig;
    elsif clk'event and clk='1' then
      if shift_sig = '1' then
        for i in 0 to 126 loop
          shiftreg_sig(i) <= shiftreg_sig(i+1);
          valid_reg(i) <= valid_reg(i+1);
        end loop;  -- i
        shiftreg_sig(127) <= shiftreg_sig(0);
        valid_reg(127) <= valid_reg(0);
      end if;
      if update_sig = '1' then
        shiftreg_sig <= query_sig;
        valid_reg <= valid_sig;
      end if;
    end if;
  end process shift_p ;

  FSM_p: process (clk, reset, state, start, query_sig, valid_sig)
  begin  -- process FSM_p
 if reset='1' then
      state <= init;
    elsif clk'event and clk='1' then
      case state is
           when init => start_sig <= '1';
                        shift_sig <= '0';
                        update_sig <= '0';
                        reg <= (others => '0');
                        state <= start_1;
           when start_1 => finish_sig <= '0';
                           start_sig <= '0';
                           for i in 0 to 126 loop
                             if start(i)='1' then
                               reg(i) <= '1';
                             end if;
                           end loop;  -- i
                           if reg = (reg'Range => '1')  then
                             update_sig <= '1';
                             state <= mux;
                           end if;
           when mux => update_sig <= '0';
                       shift_sig <= '1';
                       state <= mux2;
           when mux2 => update_sig <= '0';
                        if valid_reg(0) = '1' then
                          shift_sig <= '0';
                          state <= stop;
                        else
                          shift_sig <= '1';
                        end if;
           when stop => shift_sig <= '0';
```

```
                        finish_sig <= '1';
                        reg <= (others => '0');
                        state <= start_1;
           when others => state <= start_1;
      end case ;
  end if;
end process FSM_p;
```

B

Completeness and time complexity of the synthesizable CHR

For a demonstration of the Turing-completeness of the synthesizable CHR we refer to the studies of the computational power and complexities of CHR [96] and in particular [95]. The following resumes the most important passages. To ease the comprehension in Figure B.1 it is reported the CHR code for simulating a RAM machine. The RAM program is encoded as a sequence of `prog/5` constraints: the arguments are respectively the program line number, the next program line number, the instruction and the optional operand for the instruction. For example the `add`

Figure B.1: CHR implementation of a RAM simulator

```
prog(L,L1,const,B,A) \ m(A,_), pc(L) <=> m(A,B), pc(L1).
prog(L,L1,add,B,A), m(B, Y) \ m(A,_), pc(L) <=>
                    Z is X+Y, m(A,Z), pc(L1).
prog(L,L1,sub,B,A), m(B, Y) \ m(A,_), pc(L) <=>
                    Z is X-Y, m(A,Z), pc(L1).
prog(L,L1,mult,B,A), m(B, Y) \ m(A,_), pc(L) <=>
                    Z is X*Y, m(A,Z), pc(L1).
prog(L,L1,div,B,A), m(B, Y) \ m(A,_), pc(L) <=>
                    Z is X//Y, m(A,Z), pc(L1).
prog(L,L1,move,B,A), m(B, X) \ m(A,_), pc(L) <=> m(A,X), pc(L1).
prog(L,L1,i_move,B,A), m(B, C), m(C, X) \ m(A,_), pc(L) <=>
                    m(A,X), pc(L1).
prog(L,L1,move_i,B,A), m(B, X), m(A,C) \ m(C,_), pc(L) <=>
                    m(C,X), pc(L1).
prog(L,L1,jump,A,_) \ pc(L) <=> pc(A).
prog(L,L1,cjump,R,A), m(R, 0) \ pc(L) <=> pc(A).
prog(L,L1,cjump,R,A), m(R, X) \ pc(L) <=> X=\=0 | pc(A).
prog(L,L1,halt,_,_) \ pc(L) <=> true.
```

instruction has as operand the address of the memory cell that contains the value to be added and the address of the memory cell that has to be incremented. The memory cells are represented by `m/2` constraints that specify the address and the content of the cell. `pc/1` is the program line counter that should be initialized with the first line of the program. The performed instructions are: `const` that sets the value of a memory cell to a certain value; `add`, `sub`, `mult` and `div` that perform the corresponding arithmetic operation (using *built-in*) on a couple of cells; `move`, `i_move` and `move_i` that set the value of a given cell to the value of another cell; `jump` and `cjump` that set the program line counter value; and `halt` that stops the program execution.

It is worth noting that all the rules present in Table B.1 are synthesizable since they just introduce constraints that are previously removed. Hence a RAM machine can be simulated by the CHR synthesizable subset. The opposite direction is clearly valid as well since it is demonstrated by the numerous existing CHR implementations. The Turing-completeness of a programming language is obtained if any program can be simulated on a Turing machine and every Turing machine can be simulated by that language. Since we know the Turing-completeness of a RAM machine [7] we can easily conclude the Turing-completeness of the CHR synthesizable subset.

For an evaluation of the time complexities we refer again to [96]. The presented RAM simulator introduces at most three constraints every time a rule fires, hence the number of CHR steps is bounded by four times the number of RAM simulator steps. It follows that the CHR synthesizable subset is enough to simulate a RAM machine with the same time complexity. For the opposite direction the complexity depends on the steps required to simulate an *Introduce* transition (linear) and an *Apply* transition (T^{m-1} where T is the CHR complexity and m the maximum number of head constraints). The resulting complexity of a CHR program, with complexity T, simulated by a RAM machine is $O(T^m)$. From literature it is known that a RAM machine can simulate a Turing machine with time complexity T in $O(T)$ time while for the opposite direction $O(T^8)$ time is needed. Considering these two steps of simulation we can conclude that the CHR synthesizable subset can be simulated in a Turing machine in $O(T)$ time while we already know that the full CHR and hence the synthesizable subset can be simulated in a Turing machine in $O(T^{8m})$ time.

As final remarks we want to mention the restriction that imposes the use of ground CHR. As pointed out in [94] the implementation of the RAM machine in Table B.1 employs just ground CHR. Thus such the limitation does not affect the Turing-completeness of the synthesizable CHR as well.

Bibliography

[1] Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In G. Smolka, editor, *CP '97: Proc. Third Intl. Conf. Principles and Practice of Constraint Programming*, volume 1330, pages 252–266, 1997.

[2] Slim Abdennadher and Thom Frhwirth. Operational equivalence of chr programs and constraints. In *In 5th International Conference on Principles and Practice of Constraint Programming, CP'99, LNCS 1713*, pages 43–57. Springer-Verlag, 1999.

[3] Slim Abdennadher and Thom Fruhwirth. On completion of constraint handling rules. In *Constraint Programming: Basics and Trends, LNCS 910*, pages 90–107. Springer-Verlag, 1998.

[4] Slim Abdennadher, Thom Frühwirth, and Holger Meuss. On confluence of Constraint Handling Rules. In *CP '96: Proc. Second Intl. Conf. Principles and Practice of Constraint Programming*, volume 1118, pages 1–15, August 1996.

[5] Slim Abdennadher and Thom W. Frühwirth. Integration and optimization of rule-based constraint solvers. In *LOPSTR*, pages 198–213, 2003.

[6] Slim Abdennadher and Michael Marte. University course timetabling using constraint handling rules. *Journal of Applied Artificial Intelligence*, 14:311–325, 2000.

[7] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[8] *Stratix Series FPGA Fracturable Look-Up Table Logic Structure*, 2009. Altera Corporation.

[9] M. Arias-Estrada and E. Rodrguez-Palacios. An FPGA Co-processor for Real-Time Visual Tracking. *Lecture Notes in Computer Science*, 2438:73–98, 2002.

[10] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. *ACM SIGPLAN Notices - OOPSLA '10*, 45(10):89–108, 2010.

[11] *AutoESL DesignTechnologies INC.* http://www.autoesl.com/.

[12] Frédéric Benhamou. Interval constraint logic programming. In *Constraint Programming*, pages 1–21, 1994.

[13] A. Benkrid and K. Benkrid. HIDE+: A Logic Based Hardware Development Environment. *Engineering Letters*, 16(3), 2008.

[14] K. Benkrid and D. Crookes. From Application Description to Hardware in Seconds: A Logic-Based Approach to Bridging the Gap. *IEEE Transaction on VLSI System*, 12(4):420–436, 2004.

[15] Hariolf Betz, Frank Raiser, and Thom W. Frühwirth. A complete and terminating execution model for constraint handling rules. *Theory and Practice of Logic Programming*, 10(4-6):597–610, 2010.

[16] J. Beuchat, N. Brisebarre, J. Detrey, and E. Okamoto. Arithmetic Operators for Pairing-Based Cryptography. *Lecture Notes in Computer Science*, 4727:239–255, 2007.

[17] Stefano Bistarelli, Thom Frühwirth, and Michael Marte. Soft constraint propagation and solving in chrs. In *Proceedings of the 2002 ACM symposium on Applied computing*, SAC '02, pages 1–5. ACM, 2002.

[18] P. Bjesse, K. L. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Proc. of the International Conference on Functional Programming*, pages 174–184. ACM Press, 1999.

[19] R. Bloem, S. Galler, B. Jobstman, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from psl. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 190, 2007.

[20] T. H. Brus, C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. Clean: A language for functional graph rewriting. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 364–384, London, UK, 1987. Springer-Verlag.

[21] Joo M.P. Cardoso and Pedro C. Diniz. *Compilation Techniques for Reconfigurable Architectures.* Springer Publishing Company, Incorporated, 1 edition, 2008.

[22] Manuel Carro, José F. Morales, Henk L. Muller, G. Puebla, and M. Hermenegildo. High-level languages for small devices: a case study. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 271–281, New York, NY, USA, 2006. ACM.

[23] *Catapult C Synthesis.* Mentor Graphics.

[24] Francky Catthoor, M. Van Swaalj, Jan Rosseel, and Hugo De Man. Array design methodologies for real-time signal processing in the cathedral-iv synthesis environment. In *Algorithms and Parallel VLSI Architectures'91*, pages 211–222, 1991.

[25] Deming Chen, Jason Cong, and Peichen Pan. Fpga design automation: A survey. *Foundations and Trends in Electronic Design Automation*, 1(3):195–330, November 2006. Modern survey of FPGA CAD algorithms.

[26] Henning Christiansen. Chr grammars. *TPLP*, 5(4-5):467–501, 2005.

[27] R.C. Cofer and B.F. Harding. *Rapid system prototyping with FPGAs*. Embedded technology series. Elsevier/Newnes, 2006.

[28] Alain Colmerauer. The birth of prolog. In *III, CACM Vol.33, No7*, pages 37–52, 1993.

[29] Chris Conger and A Gordon-Ross. Fpga design framework for partial run-time reconfiguration. *Proc of 2008 International Conference on*, pages 1–7, 2008.

[30] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[31] *AutoESL DesignTechnologies INC.* NEC.

[32] D.Y. Deng, D. Lo, G. Malysa, S. Schneider, and G.E. Suh. Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric. In 43^{rd} *Annual IEEE/ACM International Symposium on Microarchitecture*, pages 137–148. IEEE, 2010.

[33] Harald Devos, Kristof Beyls, Mark Christiaens, Jan Van Campenhout, Erik H. D'Hollander, and Dirk Stroobandt. Finding and applying loop transformations for generating optimized fpga implementations. *Transactions on High Performance Embedded Architectures and Compilers I*, 4050:159–178, 7 2007.

[34] R. Domer, A. Gerstlauer, and D. Gajski. *SpecC Language Reference Manual*, 2002.

[35] Y. Dotan and B. Arazi. Concurrent Logic Programming as a Hardware Description Tool. *IEEE Transaction on Computers*, 39(1):72–88, 1990.

[36] G. J. Duck, P. J. Stuckey, M. García de la Banda, and C. Holzbaur. The refined operational semantics of Constraint Handling Rules. *Lecture Notes in Computer Science*, 3132:269–304, 2004.

[37] Gregory J. Duck, Peter J. Stuckey, Maria Garcia de la Banda, and Christian Holzbaur. Extending arbitrary solvers with constraint handling rules. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, PPDP '03, pages 79–90, New York, NY, USA, 2003. ACM.

[38] R. Kent Dybvig. *The Scheme Programming Language.* MIT Press, fourth edition, 2009.

[39] Stephen A. Edwards. The challenges of hardware synthesis from c-like languages. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, pages 66–67, Washington, DC, USA, 2005. IEEE Computer Society.

[40] E. El-Araby, P. Nosum, and T. El-Ghazawi. Productivity of High-Level Languages on Reconfigurable Computers: An HPC Perspective. In *IEEE International Conference on Field-Programmable Technology.* IEEE, 2007.

[41] Bryan H. Fletcher. Fpga embedded processor. *Proceedings of the Embedded System Conference, San Francisco*, pages 1–18, 2005.

[42] R.W. Floyd and J. D. Ullman. The compilation of regular expressions into integrated circuits. *J. ACM*, 29:603–622, 1982.

[43] Charles Forgy and John P. McDermott. Ops, a domain-independent production system language. In *IJCAI*, pages 933–939, 1977.

[44] H. Foster, Y. Wolfshal, E. Marschner, and I. W. Group. *IEEE standard for property specification language PSL*, 2005.

[45] T. Frühwirth. Parallelizing Union-Find in Constraint Handling Rules Using Confluence Analysis. In *Proc. of the 21st Conference on Logic Programming (ICLP)*, volume 3668 of *LNCS*, pages 113–127. Springer, 2005.

[46] T. Frühwirth. *Constraint Handling Rules.* Cambridge University Press, 2009.

[47] T. Fruhwirth, P. Brisset, and J.-R. Molwitz. Planning cordless business communication systems. *Expert, IEEE [see also IEEE Intelligent Systems and Their Applications]*, 11(1):50–55, 1996.

[48] Thom Frühwirth. Constraint Handling Rules. In A. Podelski, editor, *Constraint Programming: Basic and Trends — Selected Papers of the 22nd Spring School in Theoretical Computer Sciences, May 16–20, 1994*, volume 910, pages 90–107, 1995.

[49] Thom W. Frühwirth and Slim Abdennadher. *Essentials of constraint programming.* Cognitive Technologies. Springer, 2003.

[50] C. Giarratano and G. Riley. *Expert Systems: Principles and Programming.* Brooks/Cole Publishing Co, 1989.

[51] R. Gupta, G. Berry, R. Chandra, D. Gajski, K. Konigsfeld, P. Schaumont, and I. Verbauhede. The next HDL: if C++ is the answer, what was the question? In *Proc. of the 38th Conference on Design automation*, pages 71–72. ACM Press, 2001.

[52] R. K. Gupta and G. De Micheli. Hardware/Software Cosynthesis for Digital Systems. *IEEE Design & Test of Computers*, 10(3):29–41, 1993.

[53] S. Gupta, R. Gupta, N.D. Dutt, and A. Nicolau. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits.* Kluwer Academic Publishers, 2004.

[54] Sumit Gupta, Nikil D. Dutt, and Rajesh Gupta. *Spark: : A Parallelizing Approach to the High-Level Synthesis of Digital Circuits.* Springer, June 2004.

[55] *Handel-C Language Reference Manual*, 1998. Embedded Solutions Limited.

[56] Scott Hauck and Andre DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[57] Pascal Van Hentenryck and Vijay Saraswat. Constraint programming: Strategic directions. *Constraints*, 2:7–33, April 1997.

[58] Ernest Friedman Hill. *Jess in Action: Java Rule-Based Systems.* Manning Publications Co., Greenwich, CT, USA, 2003.

[59] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, August 1978.

[60] Christian Holzbaur and Thom Frühwirth. A prolog constraint handling rules compiler and runtime system. *Applied Artificial Intelligence*, 14(4):369–388, 2000.

[61] G. Hu, S. Ren, and X. Wang. A comparison of C/C++-based Software/Hardware Co-design Description Languages. In *Proc. of the 9th International Conference for Young Computer Scientists*, pages 1030–1034. IEEE, 2008.

[62] *Impulse C, Impulse Accelerated Technologies, Inc.* http://www.impulsec.com/.

[63] G. Jones and M. Sheeran. Circuit design in Ruby. *Formal Methods for VLSI Design*, pages 13–70, 1990.

[64] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. & Dev.*, 49(4/5):589–604, 2005.

[65] David Ku and Giovanni De Micheli. HardwareC - A language for hardware design (version 2.0). Technical report, 1990.

[66] I Kuon and J Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.

[67] Ian Kuon, Russell Tessier, and Jonathan Rose. Fpga architecture: Survey and challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2007. Modern survey of FPGA Architecture.

[68] Edmund S. L. Lam and Martin Sulzmann. Concurrent goal-based execution of constraint handling rules. *CoRR*, abs/1006.3039, 2010.

[69] J. Launchbury, J. R. Lewis, and B. Cook. On embedding a microarchitectural design language within Haskell. In *Proc. of the fourth International Conference on Functional Programming*, pages 60–69. ACM Press, 1999.

[70] C.Y. Lee. An algorithm for path connections and its applications. *IRE Trans. EC-10*, pages 346–365, 1961.

[71] Vladimir Lifschitz. Foundations of logic programming. In *Principles of Knowledge Representation*, pages 23–37. MIT Press, 1996.

[72] John W. Lloyd. *Foundations of logic programming.* Springer-Verlag New York, Inc., 1987.

[73] John W. Lloyd. Practical advtanages of declarative programming. In *GULP-PRODE (1)*, pages 18–30, 1994.

[74] W. Luk and S.W. McKeever. Pebble: a language for parametrised and reconfigurable hardware design. In *Field-Programmable Logic and Applications*, volume 1482 of *LNCS*, pages 9–18. Springer, 1998.

[75] Bruce J. MacLennan. *Functional programming: practice and theory.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[76] Michael J. Maher. Logic semantics for a class of committed-choice programs. In *ICLP*, pages 858–876, 1987.

[77] S. Mak and K. Lam. Serial-Parallel Tradeoff Analysis Of All-Pairs Shortest Path Algorithms In Reconfigurable Computing. In *Proc. of the IEEE International Conference on Field-Programmable Technology*, pages 302–305, 2002.

[78] Simon Marlow. *Haskell 2010 Language Report*, 2010.

[79] Marc Meister. Fine-grained Parallel Implementation of the Preflow-Push Algorithm in CHR. In *Proc. of 20th Workshop on (Constraint) Logic Programming*, pages 172–181, 2006.

[80] R. Mueller, J. Teubner, and G. Alonso. Data Processing on FPGAs. *Proceedings of the VLDB Endowment*, 2(1):910–921, 2009.

[81] Y. Oddos, K. Morin-Allory, and D. Borrione. Synthorus: Highly efficient automatic synthesis from psl to hdl. In *Proc. IFIP/IEEE International Conference On Very Large Scale Integration (VLSI-SoC'09)*, 2009.

[82] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–999, 2008.

[83] Star Galaxy Publishing. *A SystemC Primer*. Star Galaxy Publishing, 2010.

[84] F. Raiser and T. Frühwirth. Exhaustive parallel rewriting with multiple removals. In *Proc. of 24th Workshop on (Constraint) Logic Programming*, 2010.

[85] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[86] A. L. Sangiovanni-Vincentelli. Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design. *Proceedings of the IEEE*, 95(3):467–506, 2007.

[87] Vijay Saraswat. *Concurrent Constraint Programming*. MIT Press, Cambridge, U.S.A., 1993.

[88] B. Sarna-Starosta and C. R. Ramakrishnan. Compiling constraint handling rules for efficient tabled evaluation. In *Proc. of PADL '07. LNCS*, volume 4354. Springer, 2007.

[89] T. Schrijvers and B. Demoen. The K.U.Leuven CHR System: Implementation and Application. In *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, 2004.

[90] Tom Schrijvers, Bart Demoen, Gregory Duck, Peter Stuckey, and Thom Frühwirth. Automatic implication checking for chr constraints. *Electron. Notes Theor. Comput. Sci.*, 147:93–111, January 2006.

[91] A. Seawright and F. Brewer. Clairvoyant: A synthesis system for production-based specification. *IEEE Trans. on VLSI*, pages 172–185, 1994.

[92] C. Sechen. *VLSI Placement and Global Routing Using Simulated Annealing*. Springer, Boston, MA, USA, 1988.

[93] M. Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):11351158, 2005.

[94] Jon Sneyers. Optimizing compilation and computational complexity of constraint handling rules. Technical report, 2008.

[95] Jon Sneyers. Turing-complete subclasses of chr. In *Proceedings of the 24th International Conference on Logic Programming*, ICLP '08, pages 759–763, Berlin, Heidelberg, 2008. Springer-Verlag.

[96] Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of constraint handling rules. In *In Second Workshop on Constraint Handling Rules, at ICLP05*, pages 3–17, 2005.

[97] Jon Sneyers, Tom Schrijvers, and Bart Demoen. Dijkstra's algorithm with fibonacci heaps: An executable description in chr. In *WLP'06*, pages 182–191, 2006.

[98] D. Soderman and Y. Panchul. Implementing c algorithms in reconfigurable hardware using c2verilog. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:339, 1998.

[99] Charles E. Stroud, Ronald R. Munoz, and David A. Pierce. Behavioral model synthesis with cones. *IEEE Des. Test*, 5:22–30, May 1988.

[100] M. Sulzmann and E. S. L. Lam. Parallel Execution of Multi Set Constraint Rewrite Rules. In *Proc. of the 10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 20–31. ACM Press, 2008.

[101] N. Suzuki. Concurrent Prolog as an efficient VLSI design language. *IEEE Comput. Mag.*, 18(2):33–40, 1985.

[102] *Open SystemC Initiative, SystemC version 2.0 Users's Guide*, 2001. Technical report.

[103] *SytemVerilog 3.1 - Accelleras Extensions to Verilog(R)*, 2003. Accellera Organization Inc.

[104] Michael Thielscher. Flux: A logic programming method for reasoning agents. *Theory Pract. Log. Program.*, 5:533–565, July 2005.

[105] A. Triossi. Boosting chr through hardware acceleration. In *Proc. of 8th Workshop on Constraint Handling Rules*, pages 1–3, 2011. Invited talk.

[106] A. Triossi, S. Orlando, A. Raffaetà, F. Raiser, and T. Frühwirth. Constraint-based hardware synthesis. In *Proc. of 24th Workshop on (Constraint) Logic Programming*, pages 119–130, 2010.

[107] T. Uehara and N. Kawato. Logic circuit synthesis using Prolog. *New Generation Computing*, 1(2):187–193, 1983.

[108] *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*, 1996. http://www.ieee.org/.

[109] *IEEE Standard VHDL Language Reference Manual*, 1994. http://www.ieee.org/.

[110] *Virtex-6 Family Overview*, 2009. Xilinx Incorporated.

[111] W. Wayne. *FPGA-Based System Design.* Pearson Education, 2004.

[112] J. Wielemaker. An overview of the SWI-Prolog Programming Environment. In *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, 2003.

[113] W. Wolf. *Computers as Components.* Morgan Kaufmann, 2^{nd} edition, 2008.

[114] Hongwei Zhu, Stuart E. Madnick, and Michael Siegel. Reasoning about temporal context using ontology and abductive constraint logic programming. In Hans Jrgen Ohlbach and Sebastian Schaffert, editors, *PPSWR*, volume 3208 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2004.