High-Level-Synthesis by Constraint Logic Programming

Ingolf Markhof

Universität Dortmund
Lehrstuhl Informatik XII
D-44221 Dortmund
F.R. Germany

markhof@ls12.informatik.uni-dortmund.de

January 10, 1994

Abstract: Integer programming has become popular to synthesis since it allows to *compute* optimal solutions by efficient formal methods. The drawback of this approach to synthesis is its resticted mathematical model. We adopted the basic idea of handling the synthesis problem as a constraint satisfaction problem and focus on solving it by constraint search. We use constraint logic programming, which is more flexible with repect to the representation of constrains.

1. Introduction

High-level-synthesis can be decomposed into a number of distinct but not independent subtasks, i.e. scheduling, allocation and binding. Solving a set of interdependent tasks requires accurate handling of constraints, because each (partial) solution of one task constraints the set of solutions for dependent tasks.

Nevertheless, due to the complexity of the overall transformation, most high-level synthesis systems perform the subtasks of high-level synthesis (more or less) in sequential order, often by utilizing some heuristics [MCP90]. In general, the resulting designs are thus suboptimal. A different approach to high-level-synthesis which tries to avoid these drawbacks is integer programming (IP): The high-level synthesis problem is mapped to a mathematical IP model, i.e., to a set of linear (in-) equations which are to be solved in respect to a linear cost function. In this way, dependent subproblems are solved *simultaneously* and the obtained solution is *optimal* in respect to the model. [LM94] presents an integer programming approach to high-level-synthesis which subsumes the ideas of [GE91],[RJL92] and [Mar90].

While traditional synthesis systems fail to deal with constraints accurately, the drawback of the IP approach is that the domain of high-level synthesis must be mapped to a very restriced mathematical model. The formulation of the integer program is cumbersome even for simple high-level synthesis models and it's probably impossible to represent all aspects of synthesis by linear equations. The resulting IP models get very complex for enhanced synthesis models which causes serious run-time problems.

Nevertheless, surprising results have been achieved by using the IP approach. We adopted the basic idea of handling the synthesis problem as a constraint satisfaction problem and focus on solving it by constraint search. This is done by constraint logic programming (CLP) in ECL^{i}PS^e, the ECRC Common Logic Programming System [Eur93].

ECL^{i}PS^e is a based on Prolog. Programming in Prolog reduces the gap between the formal problem specification and its implementation. Prolog also supports rapid prototyping and simplifies software-maintenance. Additionally, ECL^{i}PS^e provides flexible built-in features for constraint handling. Constraints that must be represented by complex sums in the IP approach can be represented simply by a single term in ECL^{i}PS^e. There is also no restriction to represent constraints by linear equations, only. Thus it is possible to utilize constraints that cannot represented by integer programming.

Of course, it is not possible to reduce the inherent complexity of the synthesis task. Although efficient mathematical methods are used to compute a solution in the IP approach, runtime is one of its main limitations. This is also true for the CLP approach. Nevertheless, CLP is more flexible in respect to the representation of constraints and it also allows user interaction.

2. Constraint Logic Programming

In logic programming [CM87], there is no semantic attached to the primitive objects of the language. An unbound variable may match any term. Thus, any semantic of an application domain must be coded as uninterpreted structures. In contrast, the key aspect of constraint logic programming is to introduce interpreted objects, namely variables with an attached domain [FHK+93]. Relations between these variables or between expressions containing such variables are called *constraints*. A *solution* of a set of constraints is a binding of the related variables that fulfills all constraints in the set. A set of constraints is *consistent*, if there is a common solution for all constraints in the set; otherwise it is *inconsistent*. By *solving* a set of constraints we mean to find a solution for the constraint set or to detect that the constraint set is inconsistent.

In traditional logic programming the only constraint is equality between terms and the unification algorithm is used to solve these constraints. As constraints of the application domain must be coded as uninterpreted structures, inconsistency of a set of constraints will be detected only by failing to find a consistent binding. This means needless backtracking.

In constraint logic programming the unification is enhanced by a decision procedure for constraints, which is used to cut the search space: The decision procedure performs *consistency checking*, i.e. it detects inconsistentent partial solutions. It also binds variables whenever their value is fixed by the constraint set. In ECL^{i}PS^e the enhancement of traditional unification is based on two new concepts, namely metaterms and delayed goals:

A *metaterm* is a variable with an attached attribute (a term) to annotate additional semantics (notation: *Variable*{*attribute*}). Whenever a metaterm is bound, the system raises an event and the decision procedure is called. Thus, X{∈[1,2,3,4,5]}=6 will cause a *fail* immediately. Additionally, predefined predicates are provided for interpretation and manipulation of metaterms, e.g., X{∈[1,2,3,4,5]}, X #< 3 will result in X{∈[1,2]}.

A *delayed goal* is a goal (in the sense of logic programming) with unbound variables or metaterms that is not solved immediately. In respect to the conjunction of premises a delayed goal is handled as a true premise without actually executing the goal. Once the unbounded variables of the delayed goal are bound, the goal is woken up and executed at this time.

In ECL^{i}PS^e meta terms introduce interpreted objects and delayed goals are used to ensure that the metaterms are not interpreted by the unification algorithm to avoid backtracking.

3. Synthesis Model

3.1 Basic definitions

To denote the synthesis task, we use the following sets:

O	The set of operations in the dataflow
T	The set of funtional units types
I	The set of funtional units instances
F	The set of operation types
C	The set of control steps

All these sets are finite. We use small letters to denote members of these sets, i.e., $o \in O$. There are attributes attached to the sets elements by the following functions:

op_fkt:	$O \rightarrow F$
op_inputs:	$O \rightarrow \wp(O)$
ut_cost:	$T \rightarrow \Re$
ut_fkt:	$T \rightarrow \wp(F)$
ut_ctime:	$T \times F \rightarrow N$
ut_latency:	$T \times F \rightarrow N$
ui_type:	$I \rightarrow T$

Op_fkt(o) is the operation performed by o, op_inputs(o) is the set of operations that o is data depend to, ut_costs(t) is the cost for an functional unit of type t, ut_fkts(t) is the set of functions that t can execute, ut_ctime(t,f) is the delay (number of control steps) for f when performed by t, ut_latency(t,f) is the latency (number of control steps) of t when executing f, and ui_type(i) is the type t of i.

3.2 Design Issues

Given the sets O and T, the design task is to define the following functions

schedule:	$O \rightarrow C$
allocation:	$T \rightarrow N$
binding:	$O \rightarrow I$

with respect to the to the constraints described below while minimizing the cost function

$$\text{cost} = C1 \cdot \sum_{t \in T} (\text{ut\_cost}(t) \cdot \text{allocation}(t)) + C2 \cdot \text{schedule\_length}$$

whereas C1 and C2 are user defined constants and *schedule_length* = max {schedule(o) + ut_ctime(ui_type(binding(o)), op_fkt(o)) | $o \in O$}

The set I of unit instances is defined implicitly by the allocation (or vice versa) by the following equivalence: $\forall t \in T$: (allocation(t) = $k \Leftrightarrow$ |{ $i \in I$ | ui_type(i) = t }| = k).

3.3 Constraints

The user provides parameters for a priori design constraints as follows (parameter ⇒ constraints):

a) $CMAX \in$ N
 ⇒ $\forall o \in O$: 1 ≤ schedule(o) ≤ $CMAX$.
b) UT_MAX: $T \rightarrow$ N
 ⇒ $\forall t \in T$: allocation(t) ≤ $UT\_MAX(t)$.
c) $MAX\_COST \in \Re$
 ⇒ cost ≤ MAX_COST.

Furthermore, operations must be scheduled with respect to the data-dependencies and no functional unit instance can perform more than one operation at any time (if not pipelined):

d) Operation precedence constraints:
 $\forall\, o_1 \in O$: $\forall\, o_2 \in$ op_inputs(o):
 schedule(o_2) + ut_ctime(ui_type(binding(o_2)), op_fkt(o_2)) ≤ schedule(o_1).
e) Resource constraints:
 $\forall\, i \in I$: $\forall o_1, o_2 \in \{o \in O$ | binding(o) = i }:
 schedule(o_1) + ut_latency(ui_type(i),op_fkt(o_1)) ≤ schedule(o_2) ∨
 schedule(o_2) + ut_latency(ui_type(i),op_fkt(o_2)) ≤ schedule(o_1).

4. Status of Implementation

To test the applicability of CLP in the domain of high-level synthesis, we first implemented an experimental program for a simplified scheduling problem. The high-level-synthesis model was simplified as follows: A unit delay is assumed for all operations, i.e. each operation can be executed in one control step, and there is a one-to-one mapping between the operation types and the types of functional units, i.e. there is exactly one type of functional unit for each distinct operation type, i.e., op_fkt(o), and every functional unit is able to perform exactly one operation type. The program consists of less than 200 lines of source code, whereas 2/3 of the program text handles input and output only; the code for the scheduling itself is thus really short. A few goals are sufficient in ECL^{i}PS^{e} to handle the required constraints:

For each operation o_i, there is a domain variable Si, i.e., a meta term, that has to be bound to the control step of that operation. According to constraint a) from section 3.3, the domain of the variable is defined by the goal Si :: 1..CMax. Operation precedence constraints are simply denoted by the goals Si #< Sj ($\forall$ oj $\forall$ oi $\in$ op_inputs(oj)). By introducing these constraints, the domain of each Si is already reduced to the ASAP..ALAP interval of the according operation. Finally, no more than allocation(t) operations performing op_fkt(o) $\in$ ut_fkt(t) may be assigned to a control step c $\in$ {1..CMax}, which is represented by the constraints atmost(allocation(t), S_t, c), whereas S_t is the list of all Variables of operations with op_fkt(o) $\in$ ut_fkt(t).

A solution for this scheduling problem is computed by iteratively binding the domain variables Si in a branch-and-bound search, whereas consistency checking cares for efficiency by pruning the search space. The implementation was successfully applied to the elliptical wave filter benchmark.

We're currently implementing a synthesis tool for the full synthesis model described in section 3 by extending the experimental program described above. Some extensions are straigth forward, e.g., according to multiple possibilities to bind an operation to different types of functional units with different delays, the operation precedence constraints are now represented by (Si + Di #< Sj), whereas domain variable Di $\in$ { ut_ctime(t) | op_type(o_i) $\in$ ut_fkt(t) }.

5. Future Work

There are several possibilities to extend the synthesis model presented in section 3: Design aspects which are missing are registers, memories and connections. The time model which is simply based on control steps could also be replaced by true execution times to include chaining into the model. Next, this could be used to determine the length of a control step automatically. The current synthesis model also applies to basic blocks, only. To deal with more complex behavioural descriptions including control statements, a partitioning schema could be introduced. Due to the flexibility of CLP in respect to the representation of constraints, we believe that all these extensions are possible. Nevertheless, such an enhanced implementation would probably need too much runtime. Thus, another important direction of research will be the incorporation of user interaction, to enable the user to specify additional constraints to cut the search space.

6. Conclusions

From the background of the integer programming approach to synthesis, we adopted the basic idea of handling the synthesis problem as a constraint satisfaction problem. A formal description of the synthesis task was presented that can be solved by constraint logic programming. By implementing a first prototyp for a simplified problem, we found that constraint logic programming is very well suited to represent and solve high-level synthesis constraints.

References

[CM87] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. 3rd edition, 1987.

[Eur93] European Computer-Industry Research Center, Munic. *ECLiPSe - ECRC Common Logic Programming System*, 3.3 edition, March 1993.

[FHK+93] Thom Frühwirt, Alexander Herold, Volker Küchenhoff, Thierry Le Provost, Pierre Lim, Eric Monfroy, and Mark Wallace. Constraint logic programming - an informal introduction. Technical report, European Computer-Industry Research Centre, 1993.

[GE91] Catherine H. Gebotys and Mohamed I. Elmasry. Simultaneous scheduling and allocation for cost constrained architectural synthesis. *28th Design Automation Conference*, pages 2–7, 1991.

[LM94] Birger Landwehr and Peter Marwedel. Oscar: Optimum simulataneous scheduling, allocation and ressource binding based on integer programming. *Internal report LS Informatik XII, University of Dortmund.*

[Mar90] Peter Marwedel. Matching system component behaviour in mimola synthesis tools. In *Proceedings of the European Design Automation Conference*, 1990.

[MPC90] Michael C. McFarland, Alice C. Parker, and Raul Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, February 1990.

[RJL92] Minjoong Rim, Rajiv Jain, and Rento De Leone. Optimal allocation and binding in high-level synthesis. *Proceedings of the 29th Design Automation Conference*, pages 120–123, 1992.