Application of Constraint Programming to Digital System Design

Szymanek, Radoslaw; Gruian, Flavius; Kuchcinski, Krzysztof

Published in:
Proceedings of the Workshop on Constraint Programming for Decision and Control

1999

APPLICATION OF CONSTRAINT PROGRAMMING TO DIGITAL SYSTEMS DESIGN

RADOSLAW SZYMANEK, FLAVIUS GRUIAN,
KRZYSZTOF KUCHCINSKI

Linköping University, Dept. of Computer and Information Science, Linköping, SWEDEN, radsz@ida.liu.se

Abstract. This paper presents an application of finite domain constraint programming methods to digital system synthesis problems. The modeling methods address basic synthesis problems of high-level synthesis and system-level synthesis. Based on the presented models, the synthesis algorithms are then defined. These algorithms aim to optimize digital systems with respect to cost, power consumption, and execution time.

Keywords. digital systems design, constraint programming

1. INTRODUCTION

Current developments in the VLSI technology make it possible to build complex systems consisting of millions of components. Recently, the system-on-chip concept has been proposed which is integration of many different system parts on a single chip. This development has the potential of reducing cost while improving many design parameters, such as performance and reliability. The design process of a system-on-chip requires new design methods which can help the designer to explore the design space early in the design process.

The typical design problems which have to be solved during system synthesis include system partitioning, allocation of components, assignment of basic system parts into allocated components and scheduling [1]. These design steps are performed sequentially with possible iterations when the results are unsatisfactory. Many heterogeneous constraints have to be taken into account during the design process. In addition to performance and area constraints, very often we would like to consider memory or power consumption constraints. These constraints are usually difficult to include in automatic design methods and they have to be handled separately, which reduces the chance to reach good final results.

System synthesis can be defined as an optimization problem. The system is modeled as a set of constraints on fabrication cost, number of components, timing, and the goal is to find a solution which reduces a given cost function. For example, we would like to implement a given functionality on a number of processors while minimizing the execution time. This optimization will provide the fastest possible implementation with the available resources. Other optimization criteria, such as cost, power consumption or a combination of them, can be also considered.

This paper is organized as follows. Section 2 defines the computational model for our framework and presents the basic finite domain modeling techniques. Both high-level and system-level synthesis approaches using finite domain constraints are then presented in section 3. A discussion on modeling advanced design features, such as pipelining, chaining and conditional paths, is presented in section 4. Section 5 discusses

This work was supported in part by the Wallenberg Foundation project “Information Technology for Autonomous Aircraft” and the Foundation for Strategic Research, Integrated Electronic Systems program.

basic methods used for solving and optimizing the finite domain models. Finally, conclusions are presented in section 6.

2. BASIC MODELING TECHNIQUES

In this paper, we consider that digital systems are modeled using graphs where the nodes represent computations and the arcs data dependencies between them. The computation nodes can be either simple operations, such as additions and multiplications, or complex tasks representing, for example, signal or image processing algorithms. Fig. 1 depicts an example of such graph. Each node in this graph, (T_1, T_2, T_3, T_4, T_5, T_6), corresponds to a computation. An arc connects two nodes if and only if there is a data dependency between the corresponding nodes. There is an arc in the graph connecting a node with another node if the first node (sender) activates communication to another node (receiver). For example, the arc between nodes T_1 and T_3 models the communication between these nodes.

In general, functionality represented by a node is activated when the communication on all its input arcs took place. The graph models a single computation as a partial order of nodes' executions. The graph is acyclic but an implicit iterative computation is assumed. For example, in the graph depicted in Fig. 1, each computation starts from the execution of nodes T_1 and T_2 and finishes with execution of node T_6.

Each node has a deterministic execution time. A communication time is also deterministic. Both execution time and communication time can be decided before the model is built. Communication between nodes is allowed only at the beginning or at the end of the node execution.

2. 1. Finite Domain Constraints Model

The graph introduced above is modeled as a set of finite domain constraints imposed on nodes' ordering and implementation resources. We define first the variables which represent basic parameters of nodes and resources and then introduce the basic constraints on these variables.

A node is modeled as a 3-tuple $T = (\tau, \delta, \rho)$ where τ, δ and ρ are finite domain variables representing the activation time of the node (τ), the execution time of the node (δ), and the resource used to execute the node (ρ).

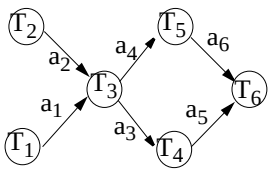

Fig. 1. An example of a computation graph

For example, for the graph depicted in Fig. 1, the following definition of the node T_1 can be made:

$T_1=(\tau_1, \delta_1, \rho_1), \tau_1::0..50, \delta_1::[2,5], \rho_1::3..4.$

Node T_1 is activated some time between 0 and 50, its execution time is either 2 or 5 time units and it uses either resource 3 or 4. For a particular implementation unique values from domains of related finite domain variables are assigned to the parameters τ, δ, and ρ. For example, T_1=(0, 2, 4).

A single node specification does not include graph information on the execution order between nodes. This is defined as inequality constraints. If there is an arc from a node T_i to a node T_j in the graph then the following inequality constraint is defined:

$\tau_i+\delta_i \le \tau_j$

Two arbitrary nodes can not, in general, be implemented on the same resource at the same time. This is usually expressed using disjunctive constraints imposed on each pair of nodes [4]. These constraints have to be defined for all nodes which can be executed in parallel, and thus we can avoid overlapping tasks' execution. It leads to creation of $\frac{n \cdot (n-1)}{2}$ constraints, in the worst case. The work presented in this paper uses CHIP 5 constrained logic programming system [1], therefore we use the global constraint `diffn/1` for this purpose. It makes use of a rectangle interpretation of nodes. A node, represented by a 3-tuple, can be interpreted as a rectangle in the time/resource space having the following coordinates ((τ_i, ρ_i), (τ_i, ρ_i+1), (τ_i+δ_i, ρ_i), (τ_i+δ_i, ρ_i+1)). Please note that we assume that each node always uses a single resource.

The `diffn/1` constraint takes as an argument a list of n-dimensional rectangles and ensures that for each pair of *i*, *j* (*i*≠*j*) of n-dimensional rectangles, there exist at least one dimension where *i* is after *j* or *j* is after *i*. The n-dimensional rectangle is defined by a tuple [O_1, ..., O_n, L_1, ..., L_n], where O_i and L_i are called, respectively, the origin and the length of the n-dimensional rectangle in the *i-th* dimension.

Using the `diffn/1` constraint we can replace the set of disjunctive constraints by imposing that all rectangles representing nodes, such as $Rect_i$=[τ_i, ρ_i, δ_i, 1] and $Rect_j$=[τ_j, ρ_j, δ_j, 1], can not overlap.

A node execution time is deterministic but it is not the same for all resources. For example, different microprocessors provide different execution time for the same task depending on the processor's clock frequency and architecture. The model makes it possible to define the node execution time as a finite domain variable capturing several execution time values. The relation between δ_i and ρ_i can be expressed using the `element/3` constraint. This constraint enforces a finite relation between the first and the third variable.

The finite relation is given by the vector of values passed as the second argument. More formally, the execution time, δ_i, for a node T_i on a resource ρ_i, is defined by the following constraint:

`element(`ρ_i`, [`T_{i1}`, `T_{i2}`, ..., `T_{iN}`], `δ_i`).`

2. 2. Redundant Constraints

The formulation presented above fully describes the graph model of the digital system and can be directly used for synthesis. However, in most of the developed applications we used redundant constraints to improve the constraint propagation. The most important redundant constraint we used is `cumulative/8`.

The `cumulative/8` constraint has been defined in CHIP to specify requirements on the tasks which need to be scheduled on a limited number of resources. It expresses the fact that, at any time instant, the corresponding total of the resources for the tasks does not exceed a given limit. The following four parameters are used: a list of the start times of tasks O_i, a list of durations D_i of tasks, a list of the amount of resources R_i required by the task and the upper limit of the amount of resources *UL*. All parameters can be either domain variables or integers. Formally, `cumulative/8` enforces the following constraint:

$$\forall i \in \left[\min_{1 \le j \le n}(O_j), \max_{1 \le j \le n}(O_j + D_j)\right] : \sum_{k\,:\,O_k \le i < O_k + D_k} R_k \le UL$$

where *n* is the number of tasks, while *min* and *max* are the minimum and maximum values in the domain of the variable respectively.

The cumulative constraint can be used to describe two types of constraints. In the first formulation O_i is replaced by τ_i, D_i by δ_i and finally R_i by 1. This models the task allocation and scheduling on the limited number of resources represented by *UL*. The second constraint represents the bin packing problem: O_i is replaced by ρ_i, D_i is always 1 and finally R_i is replaced by δ_i. The variable *UL* is constrained to the value lower or equal the execution time of the graph. Please note that the first formulation uses only τ_i and δ_i while the second one only ρ_i and δ_i. Thus they are able to offer different types of propagation.

We use also another redundant constraint `precedence/5`. This constraint takes into account, in addition to precedence constraints expressed by inequalities, the resource limitations on which jobs can be scheduled. Thus this redundant constraint gives better propagation than inequalities alone. The main limitation of this constraint is the need to express durations and resource usage as integer values.

3. DIGITAL SYSTEM SYNTHESIS

The synthesis process of a digital system starts with defining the system in an abstract way, usually by describing its functionality. This abstract representation is then refined and it becomes a description of the hardware (e.g. ASIC) and possibly some software modules, which together implement the system. We believe that the use of Constraint Logic Programming (CLP) can improve both quality and efficiency of system design steps. Below we will present a number of examples where CLP has been used for high-level and system-level synthesis addressing their different aspects.

3. 1. High-Level Synthesis

High-Level Synthesis (HLS) refers to the step in the synthesis of a digital system where a functional (behavioral) specification of a system or sub-system is transformed into a Register-Transfer Level (RTL) representation. This RTL representation will be later implemented in hardware.

During HLS one must decide the type and number of resources (adders, multipliers, ALUs, registers) needed, the right time to perform each operation, and resource which will perform it. These three problems are referred to as resource allocation, operation scheduling, and binding. Each of these problems has been proven to be NP hard. HLS targets minimization of execution time or resources cost, therefore it can be time or resource constrained.

The input to the HLS process, the functional specification, can be represented as a control data-flow graph (CDFG). A CDFG and the graph presented in Fig. 1 are very much alike. Therefore, the modeling techniques introduced in the previous section are directly applicable. The nodes in CDFG represent simple operations, such as additions, multiplications, comparisons, while the arcs describe the conditional or non-conditional data flow between different operations. For clarity we will consider here only data flow graphs (DFG), although control information can be handled as shown in [4].

The constraints described in the previous section form the basic model for the general HLS problem. Additional constraints for modeling application specific issues such as register assignment and power consumption minimization are described in the following part.

Registers are assigned to input, output, and temporary variables during high-level synthesis. To allow register sharing, the lifetimes of the variables, representing the period that the variable occupies a register, are computed and a related analysis determines register assignment. The lifetimes of the variables are modeled in our approach using rectangles which span on time axes over define-use time of the variable. This resembles a definition of variable lifetimes used in left-edge algorithm (see, for example [1]). Defining the lifetimes of variables as rectangles provides a natural way to use both `diffn/1` and `cumulative/8` constraints [5].

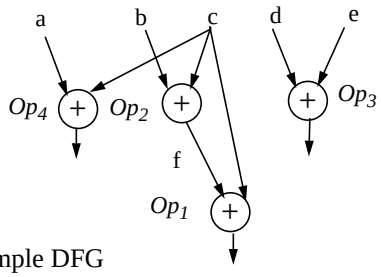

Fig. 2. A simple DFG

System power consumption is another important design issue. For CMOS digital circuits the power consumption depends mainly on the supply voltage, clock frequency, and switched capacitance:

$$P_{switching} = \alpha \cdot C \cdot f \cdot V_{dd}^2$$

Considering that the voltage and frequency are usually fixed as design requirements, and the capacitance is determined by the technological process, the only way to minimize power consumption is by minimizing switching activity α. The switching activity of a node is a measure of how much a certain node in a CMOS circuit has to switch from 1 to 0 to compute something. In other words, if the signals in a circuit are changing as little as possible during computation then the circuit will consume less power. With this observation, one could carefully schedule the order of operations on each resource such that the data is changing as little as possible at the inputs and inside the resource. Briefly, binding and scheduling influence the values and the sequence of signals applied to each resource.

Consider simple DFG shown in Fig. 2. There are several possible schedules and bindings for this graph using two adders. Each solution yields different switching activities, thus different power consumptions. Two of these possible bindings and schedules are depicted in Fig. 3. First let us consider that each operation of the DFG is executed on its own functional unit yielding a switching activity that can be calculated using signal probabilities or computed by a fast RT level simulator. Let us call this switching activity the unbounded switching ($Sw_{0,i}$) for operation Op_i. In general, during high level synthesis, several operations will be bound to the same resource determining the switching activity of the design. For example, if on a

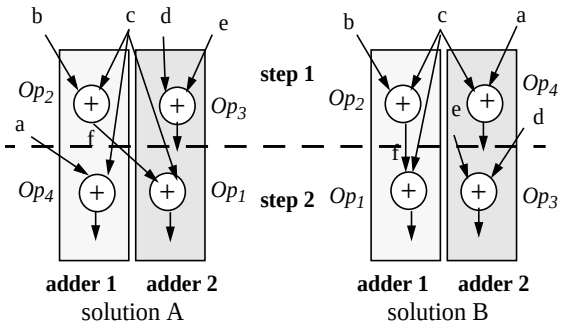

Fig. 3. Two possible schedules with different bindings for the DFG in Fig. 2

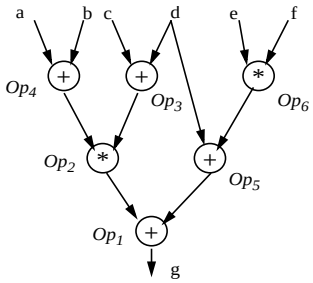

Fig. 4. Another simple DFG

certain resource operations Op_i and Op_j are executed in that order, the switching activity cannot be computed as a simple sum $Sw_{0,i} + Sw_{0,j}$ since the switching produced by Op_j is dependent on the signal values produced by the previous operation Op_i. It is closer to reality to consider switching as $Sw_{0,i} + Sw_{i,j}$ where $Sw_{i,j}$ is the relative switching between operations Op_i and Op_j. The relative switching activity describes the bit correlation of two signals and is defined as the number of different bit values of the two signals [7].

What we finally need to minimize is exactly the switching yielded by a certain sequence of operations on a certain resource. For that we have to know the sequence of operations on each resource which can be obtained in CHIP using the `cycle/n` constraint. Actually we have to deal with a slightly modified travelling salesman problem (TSP) [8] where there are as many cycles as there are resources. The nodes in the graph are the operations, Op_i, and the weights assigned to the arcs in the graph are the relative switching values. For example $Sw_{i,j}$ is the weight of the arc going from Op_i to Op_j. The unbounded switchings can be seen as arcs from a dummy node representing a resource i to a normal operation node (see Fig. 5).

For the DFG example depicted in Fig. 4, a possible design which uses three resources, two adders and one multiplier, is described in Fig. 5. The operations Op_4, Op_5, Op_1 are executed on resource *1* in this order, Op_6, Op_2 on resource *2* in this order and Op_3 on resource *3*. The switching activity is the sum of the weights of the arcs involved:

$$Sw = (Sw_{0,4} + Sw_{4,5} + Sw_{5,1}) + (Sw_{0,6} + Sw_{6,2}) + Sw_{0,3}$$

Please observe that the arcs closing the cycles, back to the dummy nodes have weight zero. In particular we used `cycle/9` to group the N operations in sets for each resource:

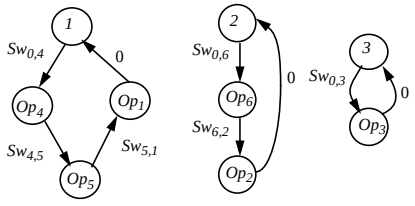

Fig. 5. Example of cycle generation for the DFG

cycle(R, [S_1, S_2, ..., S_R, S_{R+1}, ..., S_{R+N}], [0, ..., 0], MinimalCycleLength, MaximalCycleLength, [1, 2, ..., R], unused, [1, 2, ..., R, ρ_1, ρ_2, ..., ρ_N], [unused, ..., unused, τ_1, τ_2, ..., τ_N]).

where R is the number of resources used, S_i, $1 \le S_i \le N$, is the domain variable indicating an immediate successor of operation i on the specific resource, and τ_i, ρ_i are the starting time and the resource used for operation i. Having the ordering of operations on each resource it is easy to compute the overall switching activity, which is the objective function to be minimized. To extract exactly the switching values needed for the computation of this function we used an additional `cycle/11` constraint. For more information please refer to [3].

3. 2. System-Level Synthesis

Given the specification of the system functionality, the main goal of system-level synthesis is to make decisions concerning the system architecture and the system implementation on this architecture. The system functional specification is compiled into a task graph. The graph introduced in the section 2 is interpreted as a task graph where the nodes represent tasks and the arcs represent communications between them. Each task must be executed on a single processor, so for each task we need to reserve a time slot, code and data memory on the chosen processor. In our approach, we assume that there is no need for communication when two tasks are executed on the same processor since both tasks have access to the same local memory.

An architecture consists of processors and communication devices, such as busses and links. Fig. 6 depicts an example target architecture, which consists of four processors, P1, P2, P3 and P4, two links, L1 and L2, and a bus, B1.

The goal of the system-level synthesis is to find an architecture with a minimal cost, which can execute all tasks while fulfilling timing and memory constraints. The architecture is created from a set of components specified by the designer. The whole process is guided by the constraint system, which enforces the correctness of the solution by rejecting all the decisions which violate constraints.

The constraints taken into account in the presented synthesis system can be classified into two groups:

- timing constraints, and
- resource constraints.

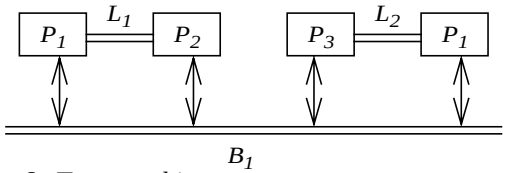

Fig. 6. Target architecture

The data dependency constraints belong to the first group and they are modeled using inequalities as presented in subsection 2.1. There are two kinds of data dependency between tasks. Indirect data dependency exists when two cooperating tasks, for example T_1 and T_3, are executed on different processors. In this case, communication a_1 depends on task T_1 and task T_3 depends on communication a_1. Direct data dependency occurs when two cooperating tasks are executed on the same processor. These two possibilities of data dependency are encoded using conditional constraints. Assuming that all data dependencies are direct, we can use a redundant constraint, `precedence/5` to get a better estimation of the lower bound for execution time of the task graph.

The problems of binding tasks to processors and communication to communication devices and scheduling them are modeled, as indicated in section 2.1, by `diffn/1` constraint. This constraint requires the task duration to be greater than zero. Since some communications can be performed in zero time, using local memory, we have to distinguish them from tasks and other communications. The way of handling “disappearing” communication is to introduce a third dimension in the `diffn/1` constraint in addition to time and resource dimensions. These communications will have different values in the third dimension. This policy ensures that non-existing communications do not restrict the solution space.

Code memory is used to store programs implementing tasks. The amount of code memory needed to implement a task depends on the processor type, but it is fixed during the execution of the whole task graph. For each processor the `sequence/5` constraint transforms the [ρ_1,ρ_2,...,ρ_m] vector, where ρ_m denotes the resource executing the m-th task, into a binary vector. A value 1 on the n-th position means that the given processor executes the n-th task and 0 otherwise. The obtained vector multiplied by the vector of code memory requirements for the given processor gives the overall usage of the code memory. This usage must not exceed the available memory.

Data memory constraint is the most complex since data memory usage changes during tasks’ execution. Data are associated with communications. First we have to allocate data memory on the processor which executes a task producing data. After transmission of the data to the processor which executes the consumer task, we need to reserve data memory on this processor until the end of the execution of the task. During transmission both processors have reserved data memory.

Example: Consider two cooperating tasks and communication between them as depicted in Fig. 7a, where T_1 is executed on processor P_1 and T_2 is executed on processor P_2. Communication C_1 is scheduled on bus B_1. The data transfer can freely

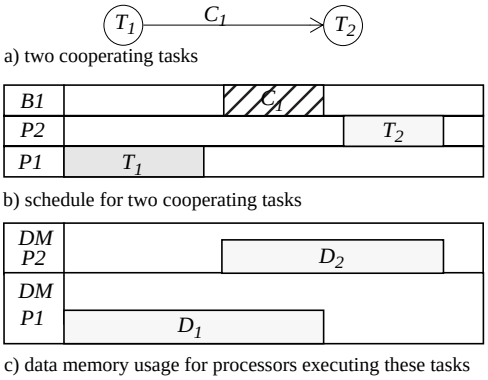

Fig. 7. Data memory requirements

occur between finishing time of T_1 and starting time of T_2, which is expressed by the following inequalities:

$$\tau_{t1} + \delta_{t1} \leq \tau_{c1} \ \wedge \tau_{c1} + \delta_{c1} \leq \tau_{t2}$$

Each communication results in two data requirements as depicted in Fig. 7c. Processor P_1 must reserve data memory, denoted by D_1, for task T_1 from τ_{t1} until $\tau_{c1}+\delta_{c1}$, where τ_{c1}, δ_{c1} denote the start time and duration of the communication respectively. Processor P_2 reserves data memory for task T_2, denoted by D_2, from τ_{c1} until $\tau_{t2}+\delta_{t2}$. D_1 and D_2 have the same height denoting the memory size.

For each processor, one cumulative constraint is created as depicted in Fig. 8. The data requirement D_1 appears in the cumulative for both processors, P_1 and P_2, because both processors can execute T_1. Task T_2 can also be executed on both processors, so D_2 exists in both cumulative constraints. Since processor P_1 executes T_1, rectangle D_1 in the cumulative constraint for processor P_1, denoted by D_1', is placed in the dotted area and rectangle D_1 in the cumulative constraint for P_2 is placed outside dotted area. The same principle applies to task T_2 and its data requirement, D2. Rectangles D_1' and D_2' are actual data requirements, while the others are not, so they are not taken into account. This is done by placing them outside the dotted area.

In addition to `cumulative/8` we have to use conditional and `element/3` constraints in order to assure that there is only one D_1' and D_2' and following equalities hold:

$$\tau_{D1'} = \tau_{T1} \wedge \tau_{D1'} + \delta_{D1'} = \tau_{C1} + \delta_{C1} \wedge \tau_{D2'} = \tau_{C1} \wedge \tau_{T2} + \delta_{T2} = \tau_{D2'} + \delta_{D2'}$$

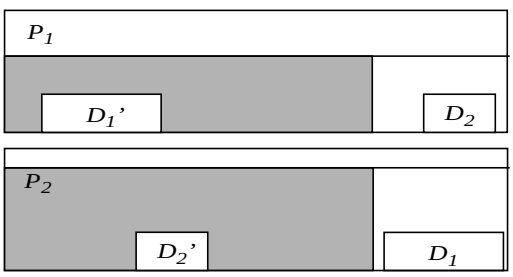

Fig. 8. Data memory constraint

when T_1 and T_2 are executed on different processors or

$$\tau_{D2'} = \tau_{D1'} + \delta_{D1'} = \tau_{T2} \wedge \tau_{T2} + \delta_{T2} = \tau_{D2'} + \delta_{D2'} \wedge \tau_{D1'} = \tau_{T1}$$

when T_1 and T_2 are executed on the same processor. Using this formulation, we can ensure that cumulative usage of data memory depicted as rectangles in the dotted area does not exceed the available data memory.

4. ADVANCED FEATURES

A number of useful extensions to the basic formulation introduced in section 2 can be defined to consider special features such as pipelined components, chaining, algorithmic pipelining and conditional execution. They are discussed in this section.

Modeling *pipelined components* can be accomplished by defining 3-dimensional rectangles, in which the third dimension represents subsequent stages of the component. For example, Fig. 9 depicts a design which uses a two stage pipelined component. The first stage, S_1, is represented by the cube of height 1 located between τ_0 and τ_1 and originated at coordinate 0 in the third dimension. The second stage, S_2, is represented by the cube of height 1 located between τ_1 and τ_2 and originated at coordinate 1 in the third dimension. All non-pipelined operations, such as the operation Op_j depicted in Fig. 9, have heights of 2 and therefore can not be placed together with neither the first, nor the second stage of the pipelined subtask. "Packing" of operations represented by 3-dimensional rectangles enables placement of the stage one and two of different operations at the same resource/time location since they do not overlap in the third dimension. Other non-pipelined operations can not collide with the pipelined ones since they have the height 2. The finite domain constraint definition for the example in Fig. 9 is the following:

$$\texttt{diffn}([[\tau_{i,S1}, \rho_i, 0, \delta_{i,S1}, 1, 1], [\tau_{i,S2}, \rho_i, 1, \delta_{i,S2}, 1, 1], [\tau_j, \rho_j, 0, \delta_j, 1, 2]]) \ \wedge \ \tau_{S1} + \delta_{S1} = \tau_{S2}.$$

This formulation can be extended into n-dimensions, if there are more different pipelined components.

Chaining refers to the high-level synthesis technique of scheduling two or more data-dependent operations into the same clock cycle. It is achieved by connecting

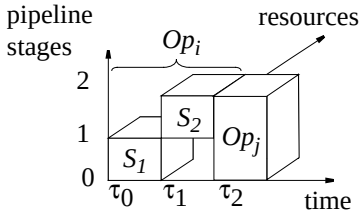

Fig. 9. Resource sharing constraints for pipelined components.

the output of one functional unit directly to the input of the following functional unit without storing a temporary value in a register. In one clock cycle the functional unit can not be reused by another operation because it still propagates results which are stored at the end of this clock cycle. This introduces additional constraints on chaining.

Fig. 10 illustrates the basic idea of modeling chaining using finite domain constraints. Three dimensional rectangles are used for this purpose. The three dimensions are used to represent resources, clock cycle and a relative position of an operation within a clock cycle, called here a step. Every clock cycle can be filled with several operations as long as they fit within the limits of the clock cycle (the rectangle boundaries). Two `diffn/1` constraints are used to impose basic requirements on the implementation. The first `diffn/1` constraint specifies the structure depicted in Fig. 10 and is defined by the following constraint:

$\texttt{diffn}([[\tau^S_i,\rho_i,\tau^C_i,\delta_i,1,1],\ldots,[\tau^S_j,\rho_j,\tau^C_j,\delta_j,1,1]]).$

The second constraint is used to forbid situations when the same resource is shared within the same clock cycle. It is defined using a projection of rectangles on the resource/clock cycle surface as a `diffn/1` constraint on two dimensional rectangles as given below.

$\texttt{diffn}([[\tau^C_i,\rho_i,1,1], \ldots, [\tau^C_j,\rho_j,1,1]]).$

The relation between previously introduced start time of an operation, τ_i, and two new parameters τ^C_i and τ^S_i is defined for every operation by the following equation:

$\tau_i = \tau^C_i * N + \tau^S_i,$

where N is the number of steps in the clock cycle.

Pipelining a data-flow graph is an efficient way of accelerating a design [4]. It introduces, in fact, new constraints on location of rectangles. This method is well known in computer architecture area, where two dimensional reservation tables are used for pipeline analysis. This approach is compatible with our methodology. Introducing an n stage pipeline of the initiation rate of k time units is equivalent to a placement of n copies of existing rectangles, starting at positions k, $2{\cdot}k$, $3{\cdot}k$, etc. This prevents placing operations in forbidden locations, which are to be used by subsequent pipeline instances. Since the operation parameters are defined by domain variables, the copies of the current rectangles do not define final operation positions but these positions will be adjusted during an assignment of values to domain variables.

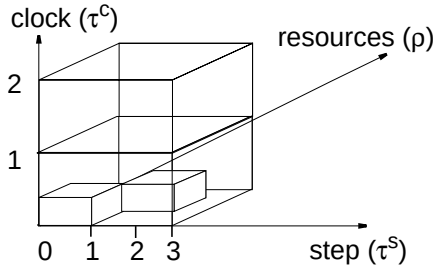

Fig. 10. Rectangle representation of chaining.

The following constraints define two-stage pipeline for two operations Op_i and Op_j, depicted in Fig. 11, with initiation rate k:

$\tau_{i,k} = \tau_i + k,\ \tau_{j,k} = \tau_j + k,$
$\texttt{diffn}([[\tau_i,\rho_i,\delta_i,1],\ [\tau_j,\rho_j,\delta_j,1],$
$[\tau_{i,k},\ \rho_i,\ \delta_i,\ 1],\ [\tau_{j,k},\ \rho_j,\ \delta_j,\ 1]]).$

The graphical, rectangle representation of these constraints is depicted in Fig. 11

The rectangle based resource constraints can be easily extended to handle *conditional* nodes. The conditional node is executed only if the conditions assigned to its input arcs are true. The value of this condition can not be statically determined and therefore we need to schedule both true and false execution cases. The presented formulation of the resource constraints which uses 2-dimensional rectangles in the time/resource space needs to be extended to cover conditional execution. The main idea of representing conditional nodes is to extend rectangles into higher dimensions. In principle, one more dimension for every new condition is used. The conditional nodes start in the third dimension either at 0 or 1, depending on the condition, and have height 1. They can share the same time/resource place since they can be placed "one on top of the other". Other computational nodes can not be placed together with conditional ones since in this formulation they have height 2.

5. OPTIMIZATION METHODS

Standard CLP optimization method is based on branch and bound (B&B) algorithm. It can be successfully applied to middle size problems, but large problems with heterogeneous constraints require more sophisticated optimization methods. The big advantage of CLP is the possibility to create new heuristics using available meta-heuristics. In our systems, we use credit search heuristic [6]. Using credit search we are

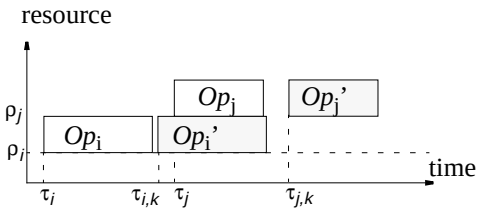

Fig. 11. A graphical representation of the resource constraint for algorithmic pipelining.

able to partially explore the whole tree and to avoid situations when the search is stuck at one part of the tree.

In this paper, we will present the heuristic invented for system-level synthesis, presented in [9], produces good results for large industrial size examples consisting of 100 tasks and 120 communications. Similar heuristics are used for other problems. The decision flow of this heuristic is following:

- Assignment of tasks to the resources
- Assignment of execution intervals to each task
- Assignment of time slots for executing tasks

First step of the heuristic is to assign tasks to processors and communication devices. The assignment tries to select the cheapest processors for a given task while minimizing the code memory usage. When the task cannot be assigned to none of the processors which are already present in the architecture than a new one is added.

In the second step we assign an execution interval for each task and communication. The number of intervals depends on the duration of the task, larger task duration gives smaller number of intervals. Tasks and communications are then divided into three groups depending on the position in the graph. For tasks which are close to the start time of the task graph the intervals with the smallest starting time are tried first. For tasks positioned in the middle of the execution period the middle intervals are selected first, and finally for tasks from the end of the execution period we assign intervals with the largest starting time. This approach allows to scatter tasks and communication evenly in time domain. The search for the correct assignment is done using credit search.

In previous step, the execution interval for each task was decided. The third step assigns the actual time slots within previously decided intervals. Since the search space is very restricted the branch and bound search is performed.

The heuristic backtracks whenever it can not find correct assignment at any step. For example, if during credit search of the intervals no correct assignment can be found then our heuristic finds a new allocation of tasks and communications to the resources and credit search is performed again. After finding a solution a new constraint is added which restrict the cost of the next solution to be smaller than just obtained and the heuristic is restarted.

6. CONCLUSIONS

In this paper, we have presented methods for digital system modeling and synthesis using finite domain constraints and CLP paradigm. We have addressed both high-level and system-level synthesis targeting different optimization goals. First, the basic formulation of the computation graphs has been introduced and formalized using finite domain constraints. Then we have shown how to use this formulation together with different extensions for high-level synthesis. The presented methods make it possible to optimize design's execution time, resources and power consumption. System-level synthesis has been defined in a similar way but it was extended with important code and data memory constraints. The introduced modeling techniques have been later used for synthesis by applying optimization methods based on B&B algorithms and domain specific heuristics.

Extensive experiments have been carried out. The experimental results presented in [3, 4, 5, 9] prove the usability of the proposed methods for large scale designs which contain up to ~200 computational and communication tasks. They show that the CLP with finite domain constraints and particularly the CHIP system provide a good base for solving many problems from the area of digital system design which require combinatorial optimization methods. These methods are specially well suited for the cases when many heterogeneous constraints are required for the problem specification.

7. REFERENCES

1. Eles P., Kuchcinski K. and Peng Z.: System Synthesis with VHDL, Kluwer Academic Publisher, 1997
2. CHIP, System Documentation, COSYTEC, 1996
3. Gruian F. and Kuchcinski K.: Operation Binding and Scheduling for Low Power Using Constraint Logic Programming, Proc. 24th Euromicro Conference, Workshop on Digital System Design, Västerås, Sweden, August 25-27, 1998
4. Kuchcinski K.: Embedded System Synthesis by Timing Constraints Solving, Proc. of the 10th Int. Symposium on System Synthesis, Sep. 17-19, 1997, Antwerp, Belgium
5. Kuchcinski K.: An Approach to High-Level Synthesis Using Constraint Logic Programming, Proc. 24th Euromicro Conference, Workshop on Digital System Design, Västerås, Sweden, August 25-27, 1998
6. Beldiceanu N., Bourreau E., Simonis H. and Chan P.: Partial search strategy in CHIP, Presented at 2nd Metaheuristic International Conference MIC97, Sophia Antipolis, France, 21-24 July 1997
7. Raghunathan A., Jha N. K. : Behavioral Synthesis for Low Power, Proceedings of ICCD 1994
8. Reeves C. R.: Modern Heuristic Techniques for Combinatorial Problems, Blackwell Scientific Publications, 1993
9. Szymanek R., Kuchcinski K.: Design Space Exploration in System Level Synthesis under Memory Constraints, 25th Euromicro Conference, Workshop on Digital System Design, Milan, Italy, September 8-10, 1999