Integrating formal, soft and diagrammatic approaches in high-level synthesis and hardware-software co-design[⋆]

András Orbán Zoltán Ádám Mann

Budapest University of Technology and Economics
`{inti,manusz}@cs.bme.hu`

Abstract

In this paper, preliminary results and research directions in high-level synthesis and hardware-software co-design are presented. The main methods are demonstrated on two case studies. The first one shows the usage of formal and soft methods (application of graph theory, constraint logic programming and a genetic algorithm) on the scheduling problem of high-level synthesis. The second case study demonstrates the application of formal verification techniques, such as model checking and propositional provers. Both case studies show methods that are usually employed in hardware design only, but their usage in software design would also be possible. At the end of the paper, a hardware-software co-design framework is suggested that integrates all these approaches with the designer-friendly diagrammatic techniques of software design.

Keywords: *hardware-software co-design; high-level synthesis; formal, soft and diagrammatic methods; scheduling; design verification*

1 Introduction

Although there are many connections and many similarities between hardware and software design, the two fields have faced somewhat different challenges and evolved in some respects quite independently. In recent years though, with the continuous need for designing systems with growing complexity, there is a tendency to place the level of abstraction of the design process as high as possible, even higher as the decision whether to realize components in hardware or software. The transformation from the high-level design to the hardware or software (or mixed) implementation should be automated as much as possible.

To address the *high-level synthesis* (HLS) problem, the PIPE system [2] was developed at the Department of Control Engineering and Information Technology of Budapest University of Technology and Economics. PIPE takes as input an Elementary Operation Graph (EOG), the nodes of which are elementary operations (EOs) – such as, for instance, additions – and the edges represent the flow of data, also specifying precedences between the operations. The output of PIPE is the scheduled and allocated EOG, *i.e.* the starting time of the EOs is determined and the EOs are mapped to physical processors. PIPE also handles pipeline systems, *i.e.* the restart time of the system can be specified and PIPE automatically makes the necessary modifications to the EOG, by inserting buffers or replicating EOs.

Although HLS was originally invented for hardware synthesis, its methodologies can also be used in the case of software systems. The transition from hardware to software is most naturally done by using the notion of 'intellectual property': an IP is basically a component with a well-defined interface, but which may be implemented in either hardware of software. Thus, the EOs are mapped to IPs instead of processors. Of course, hardware-software co-synthesis (HSCS) can be looked at from the other perspective as well, namely by designing a software system, and then realizing the performance-critical parts in hardware. But as it turns out, finding the optimal partition between hardware and software, also taking communication costs into account, is much more complex, involving NP-hard problems [2].

At this point of the transition from HLS to HSCS, the methods of software design also have to be considered. Fortunately, there is a widespread standard of diagrammatic software design, namely UML (Unified Modeling Language), that also inherits from, and extends former methodologies such as OMT. For our aims it is vital that UML also enables the formal specification of semantics using constraints expressed in OCL (Object Constraint

[⋆]This paper has been accepted for publication in the proceedings of the workshop 'Integrating Diagrammatic and Formal Specification Techniques' at Informatik 2001 (Vienna, 25-28 September 2001)

Language), because this way diagrammatic design may be integrated into an automated verification and synthesis framework.

The paper is organized as follows. Section 2 describes the scheduling and allocation problems, whereas section 3 and section 4 present the authors' results on applying formal and soft methods on the scheduling problem. Section 5 describes a case study conducted by the authors to uncover the state of the art in formal verification techniques. After presenting all these preliminary results, section 6 concludes the paper by describing how all these methods may be integrated to form a generic HSCS framework, and how the participating fields would benefit from this.

2 Scheduling and allocation in HLS

The task of the scheduler is to find an optimal valid scheduling in a given EOG. Usually, there is an ASAP (As Soon As Possible) and ALAP (As Late As Possible) value for the starting time of each EO. The scheduler must fix the starting times between ASAP and ALAP, but at the same time it has to make sure that the precedences specified by the EOG are satisfied.

Now what is understood by an optimal schedule? Actually, optimality should be measured in terms of hardware costs, *i.e.* the number of processors needed. However, it is by no means trivial to calculate the number of processors for a given schedule: this is exactly the task of allocation.

Allocation maps the EOs to physical processors. It has to make sure that concurrent operations will not be mapped to the same processor. This can be represented by another graph, the so-called concurrency graph: its nodes are again the EOs, but this time the (undirected) edges between two nodes mean that the corresponding two operations are concurrent. Obviously, allocation is equivalent to the coloring of this graph. Allocating in a minimal number of processors means finding the chromatic number of the concurrency graph, which is an *NP*-hard problem.

Although there are relatively fast heuristic allocation algorithms, they are not fast enough to be called as a subroutine perhaps thousands of times by the scheduler. Therefore, another quantity was chosen as the objective function in the scheduler, a quantity that is easier to compute but the optimality of which probably implies the optimality of the number of processors as well: the number of compatible (*i.e.* not concurrent) pairs. If the number of compatible pairs is high, this means that there are only few edges in the concurrence graph and therefore its chromatic number is probably small, so the number of necessary processors is low.

Since allocation is *NP*-hard, this implies that the joint problem of scheduling and allocation is also *NP*-hard. One of the authors' theoretic results is the proof that the scheduling problem (as defined above) is also *NP*-hard (see [1]). It is questionable whether it is useful to divide an *NP*-hard problem into two *NP*-hard problems. The reason for this decomposition is that there are significant differences between *NP*-hard problems as well, especially concerning approximate solutions. Note that in many cases it is not vital to find the absolute optimum but only a good enough solution.

In the case of the allocation problem there are indeed quite efficient heuristic algorithms. Unfortunately this is not the case with scheduling. In PIPE for instance, since the actual optimization is done in the scheduler, it is the most critical system component concerning both running time and the quality of the found solution. For big input graphs, the running time of PIPE is essentially the same as that of the scheduler. (PIPE contains a so-called force-directed scheduler [10].) This motivates the search for better and better scheduling algorithms.

In the next two sections two scheduling algorithms are presented that can replace the scheduler in PIPE. A more detailed description and a thorough evaluation of the algorithms is given in [1].

3 Genetic algorithm

The authors first applied a general heuristic – a genetic algorithm (GA [7]) – to the scheduling problem. The application involves specifying what individuals, the population, genetic operations and the fitness function are.

Actually, the scheduling problem is fortunate from the point of view of a genetic algorithm. The applicability of genetic algorithms requires that the solutions of the optimization problem can be represented by means of a vector with meaningful components. There is an obvious vector representation in the case of the scheduling problem: genes are the starting times of the EOs. The order of the genes is not indifferent either: for the efficiency of recombination it is vital that genes next to each other do represent correlative pieces of information.

Choosing the population is not that straight-forward. The question to answer is whether non-valid schedulings (*i.e.* schedulings violating some precedences defined by the EOG) should also be allowed. Since non-valid schedulings have no real physical meaning, it seems to be logical at first glance to work with valid schedulings only. Unfortunately, there are two major drawbacks to this approach. First, this may constrain efficiency severely. Namely, it may be possible to get from a valid individual to a much better valid individual through a couple of non-valid individuals, whereas it may not be possible (or perhaps only in hundreds of steps) to get to it through valid ones. In such a case, if non-valid individuals are not permitted, one would hardly arrive to the good solution. The other problem is that it is hard to guarantee that genetic operations do not generate non-valid individuals even from valid ones. This holds for both mutation and recombination.

For these reasons the authors decided to permit any individual in the population, not only valid ones. Of course the scheduler must produce a valid scheduling at the end. In order to guarantee this, there must be valid individuals in the initial population and the fitness function must be chosen in such a way that valid individuals do not become extinct. The authors' further result is a theorem for generating several valid individuals [1].

As genetic operations, mutation, recombination and selection were used. Mutation is done in the new population; each individual is chosen with the same probability. Recombination is realized as cross-over, from two individuals of the old population two new individuals are generated. The roulette method [7] is used for choosing the individuals to recombinate. Selection is realized as filling some part of the new population with the best individuals of the old population.

The fitness has two components: the first one is the actual objective function, namely the number of compatible pairs. If only valid individuals were allowed, the fitness would be equal to the objective function. But non-valid individuals are also allowed; however, they should have lower fitness values. This is why a second component of the fitness is needed. Since these individuals should be motivated to be less and less invalid, the second component of the fitness is a measure of the invalidity, namely the number of collisions, *i.e.* the number of precedence rules (edges of the EOG) that are violated. So the fitness is monotonously increasing in the number of compatible pairs and monotonously decreasing in the number of collisions.

Although optimization can be made more efficient by means of a large population, the scheduler must give only one solution at the end. However, there may be dozens of valid individuals with a high objective value in the last population. So the best valid individuals are chosen and the allocation process is run for all of them. Afterwards the best one is chosen (in terms of used processors and not compatible pairs anymore) as output.

4 CLP-based solution

As an alternative solution, the authors implemented another algorithm for the scheduling problem in HLS. It is based on Constraint Logic Programming (CLP [6]). More specifically, it uses the CLP(FD) library of SICStus Prolog, a library capable of handling finite domain variables and constraints defined on them. Contrary to the genetic algorithm presented above, this solution is fully deterministic. It makes use of a heuristic based on engineering experience, as well as the power of CLP to reach good solutions by traversing only a fraction of the search space.

The algorithm, called CCLS (Compatibility Controlled List Scheduling) is a variant of list scheduling algorithms [2]. The main idea of all list scheduling algorithms is that the nodes are traversed once and their starting time is fixed to the position that seems to be best in the given situation. The order of the nodes is determined according to a given heuristic derived from practical experience. The advantage of this method is its speed, while the major disadvantage is that it examines only a minor part of the whole state space, thus often yielding suboptimal solutions.

CCLS tries to eliminate this disadvantage but simultaneously keep the advantages by a good compromise. Instead of taking every node one by one and fixing it to its currently optimal place, small groups are formed from the nodes, and the groups are fixed to their optimal place considering the aspect of the whole group. With this change more possibilities in the search space are adverted, therefore a better result can be achieved, but the nodes are traversed still only once, so the algorithm remains reasonably fast.

Obviously, the effectiveness of the algorithm depends significantly on the size of the groups (grp). $grp = 1$ corresponds to the classical list scheduling; if grp equals the number of the nodes, then the whole state space is searched. By changing the value of grp, the effectiveness/required time ratio can be adjusted as necessary.

The objective function is the number of compatible pairs. In order to determine this number in a given state of the algorithm, every node has to be fixed, *i.e.* the starting time of each node must be exactly specified. As a consequence, the algorithm has to start from a valid schedule and change in each step the starting time of some nodes to get a better, but still valid schedule. The current implementation starts with the ALAP schedule which is guaranteed to be valid. In a general step of the algorithm, all the possible positionings of the nodes in the current group are considered, the best one is chosen, and in later steps these nodes remain unchanged. This results in the advantage that if there is not enough time to wait until the end of the algorithm, it can be interrupted at any time and it will still produce a fairly good, and valid schedule.

The biggest problem in the implementation of the outlined algorithm is that the fixation of a node can affect the mobility domain of other nodes, and these changes have to be updated continuously in every step of the algorithm. It is possible that by timing a node to another time slot, one of the precedences defined by the EOG is hurt. In order to correct this, one of its neighbors has to be moved as well, so the change may need to be propagated through the whole EOG. This is quite a difficult task in a traditional programming language like C. That is why CLP was chosen, because it makes sure that the defined constraints are not violated.

To every node a constraint variable was ordered that denotes the starting time of that node. The initial domain of these variables is the [ASAP, ALAP] interval. To define the constraint that adjacent nodes in the elementary operation graph should be run sequentially, let us assume that an edge goes from node v_i to v_j. Let V_i and V_j be the corresponding variables and let d_i be the duration of node v_i. The following constraint expresses that v_j can only be started after finishing v_i: $V_i + d_i \leq V_j$. This kind of constraint must be defined for every edge in the EOG.

The task of defining the number of compatible pairs is rather complicated, because the compatibility relation of a pair of nodes depends on several factors. The CONCHECK algorithm described in [2] had to be implemented to determine the compatibility of a pair of nodes and to order a Boolean variable to it (1 if compatible, 0 if not). The sum of these Boolean values is the number of compatible pairs which CCLS tries to maximize.

5 The design verification problem

The preceding sections described several methods for automated system synthesis provided that a high-level formal specification of the model exists. This section discusses possible techniques to formally specify software or hardware systems and afterwards verify them against several criteria.

Hardware and software systems will inevitably grow in scale and functionality. Because of this increase in complexity, the likelihood of subtle errors is becoming much greater. Moreover, some of these errors may cause catastrophic loss of money, time, or even human life. A major goal of software and hardware engineering is to enable developers to construct systems that operate reliably despite this complexity. One way of achieving this goal is by using formal methods, which are mathematically-based languages, techniques, and tools for specifying and verifying such systems. The use of formal methods does not *a priori* guarantee correctness. However, they can greatly increase our understanding of a system by revealing inconsistencies, ambiguities, and incompleteness that might otherwise go undetected.

Verification systems consist of the following three parts:

1. Framework to specify the **model**: in most cases a formal description language is available for that purpose.

2. Specification language for the **property** to verify: some kind of temporal logic is expected.

3. **Verification method** to decide whether or not the given property is true in the model.

5.1 Hardware verification

The most common technique in hardware verification is the so-called *model checking*. It is – according to the classification of verification systems – an automatic, model-based, property verification method, which is mostly applied to parallel, reactive systems after system development.

The first papers about model checking appeared at the beginning of the eighties; at that time it was possible to handle systems with a couple of thousands of states only. In the first part of the nineties a new technique called *symbolic model checking* was developed by Burch, Clarke, McMillan and Dill [4] and Berthet, Coudert and Madre [5], which in extreme cases makes it possible to deal with 10^{100} states due to an effective representation of states called *binary decision diagrams* (BDDs). Recently, an alternative approach has been proposed: the verification problem is transformed to an equivalent satisfiability instance and solved by means of highly efficient propositional provers. Representatives of that are *bounded model checking* [3] and the usage of *Boolean expression diagrams* (BEDs) instead of BDDs.

The authors compared symbolic model checking and verification with propositional provers in a case study [9]. To demonstrate the power of these methods and the wide range of their applicability, the selected benchmark problem was not from the field of hardware design: different versions of an NP-hard game, called *peg-solitaire*, were used.

Peg-solitaire is a board game in which the objective is to reach a desired final situation from a starting position of the figures by a sequence of correct steps. A step is either a horizontal or a vertical jump with an arbitrary figure over a neighboring one that must be taken off the table after the jump.

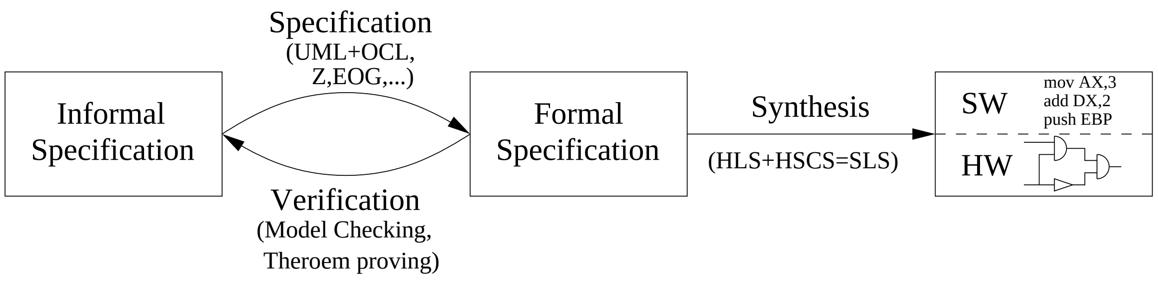

Figure 1: Vision of system engineering

Several versions of the game – varying in size and shape of the board and dealing with different initial and final situations – have been described as SMV (Symbolic Model Verifier [8]) models, and the property *it is not true that the game is solvable* has been specified in CTL (computation tree logic). The SMV system verifies the property and in the case of a negative answer it also generates a counter-example, *i.e.* it solves the game.

In the other approach the prover SATO (Satisfiability Testing Optimized [11]) has been used to test the satisfiability of a propositional formula equivalent to the solution of the game. To transform the verification problem to a satisfiability problem two kinds of variables have been introduced (the number of fields on the board is n and s is the number of steps):

- $x_{i,j}$: true, iff the j-th field is occupied in the i-th step, $0 \leq i \leq s$, $0 \leq j < n$.
- $h_{i,j,k}$: true, iff the figure on the j-th field jumps in the i-th step in direction k, $1 \leq i \leq s$, $0 \leq j < n$, $k \in \{north, east, south, west\}$.

The rules of the game can be specified with these variables, for example to express the consequences of a jump with the figure on the j-th field in the i-th step in direction *east*:

$$\begin{aligned} h_{i,j,east} \rightarrow \quad & (x_{i,j} \wedge x_{i,j+1} \wedge \neg x_{i,j+2} \wedge \\ & \neg x_{i+1,j} \wedge \neg x_{i+1,j+1} \wedge x_{i+1,j+2}) \end{aligned}$$

should be declared. With formulas similar to this, the whole behavior of the game can be described; after that it should be transformed to CNF and be given as input to SATO.

Both methods were able to solve non-trivial game-instances but the limit of their applicability was also explored with bigger examples. The calculation with BDDs makes symbolic model checking very memory-intensive in contrast to the satisfiability approach, which can be rather time-consuming. On the other hand, the results were also compared to a naive search algorithm implemented in C, based on recursive exhaustive search, which demonstrated the power of formal verification methods.

Despite their imperfection these methods proved to be useful, and in the near future they could be integrated in the automated system-development process.

5.2 Software verification

In the present software-development practice it is still rather rare to use formal specification and verification methods, although the size and complexity of current software systems would require formal proof of software correctness. The informal requirements of the system given by the procurer should first be formally specified and the specification should be verified.

Formal methods support precise and rigorous specification of those aspects of a computer system that can be expressed in the language. Since defining what a system should do and understanding the implications of these decisions, are the most troublesome problems in software engineering, the use of formal methods has major benefits. In fact, practitioners of formal methods frequently use them solely for recording precise specifications, not for formal verification.

Formal methods can deal with many areas of concern to software engineers, but have not been much used other than in research organizations. Areas in which researchers are exploring formal methods include software safety and security, fault tolerance, response time, space efficiency, reliability, human factors, and software structure dependencies.

The most commonly used specification language, UML, is not a formal language. It is a standard diagrammatic method to capture system behavior, but an UML specification can often be interpreted in many different ways. An extension of UML is OCL (Object Constraint Language) that allows the developer to define several pre- and post-conditions, invariants or guards to specific elements of the UML model. There have been some efforts to formally verify UML-OCL designs (see e.g. `http://i12www.ira.uka.de/key`).

Some of the most well-known formal methods consist of or include specification languages for recording a system's functionality. Examples of those are Z, Dynamic Logic, Communicating Sequential Processes (CSP), Larch, Formal Development Methodology (FDM). A semi-formal graphical method is DFD (Data Flow Diagram), a completely formal one is the Petri nets.

The reasons why formal methods are not yet widely used in software-development include:

- Tools for formal software specification and verification are not integrated into the industrial software engineering process
- Users of verification tools are expected to know syntax and semantics of one or more formal languages. Even worse, often a knowledge of the employed logic calculus and proof strategies is necessary.
- Formal methods can prove that an implementation satisfies a formal specification, but they cannot prove that a formal specification captures a user's intuitive informal understanding of a system.

6 Conclusion

In our vision of system engineering (see Figure 1), the system is first informally specified in a high-level language near to the mentality of the user. After that, with the aid of formal specification methods the system's behavior should be defined completely and precisely, and the generated specification should be checked against the intentions with formal verification techniques. This phase is possibly not only one single step but is rather a cyclic process of verifying the system against requirements and synthesizing system parts from the requirements. Note that until this stage of development it is not even decided which parts of the system will be realized in software and which in hardware.

After a formal model of the system is produced, the instruments of HLS and HSCS can be applied to realize an optimized implementation automatically. The integration of all these technologies leads to the so-called *System Level Synthesis*, which as indicated in its name is an even higher-order abstraction that does not distinguish between software and hardware design.

References

[1] P. Arató, Z. A. Mann, and A. Orbán. Formal methods in high-level synthesis. To be published.

[2] P. Arató, T. Visegrády, and I. Jankovits. *High level synthesis of pipelined datapaths.* John Wiley and Sons, 2001.

[3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, 1999.

[4] J. R. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[5] O. C. Berthet and J. C. Madre. New ideas on symbolic manipulations of finite state machines. In *IEEE International Conference on Computer Design*, 1990.

[6] D. Diaz. *CLP(FD), User's Manual*, 1996.

[7] W. Kinnebrock. *Optimierung mit genetischen und selektiven Algorithmen.* Oldenbourg, 1994.

[8] K. L. McMillan. *The SMV language*, 2000.

[9] A. Orbán. Analyse zweier formaler Verfikationsmethoden. Master's thesis, Universität Karlsruhe, 2001.

[10] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioural synthesis of ASICs. *IEEE Transactions on Computer Aided Design*, 1989.

[11] H. Zhang. SATO: An efficient propositional prover. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249. Springer-Verlag, 1997.