Systematic Test Case Instance Generation for the Assessment of System-level Design Space Exploration Approaches

Kai Neubauer, Christian Haubelt
University of Rostck
Germany
{kai.neubauer, christian.haubelt}@uni-rostock.de

Philipp Wanko, Torsten Schaub
University of Potsdam
Germany
{wanko, torsten}@cs.uni-potsdam.de

Abstract. The design of embedded systems gets continually more arduous as the complexity of applications and hardware platforms advance to satisfy the increasing demands on functionality, performance, and power consumption. Mostly however, the concurrent fulfillment of those demands are impossible because quality parameters are usually conflicting with each other and cannot be guaranteed simultaneously. Thus, to find the best compromises of all possible solutions, an efficient Design Space Exploration (DSE) becomes imperative. While, in recent time, many DSE techniques to the system-level synthesis problem of embedded systems design have been proposed, a systematic approach on how to produce a viable set of variant test cases with definite similar properties is not available. In this work, we therefore propose a methodology for the test case generation for DSE techniques and present a versatile and easily expendable benchmark generator based on Answer Set Programming (ASP) that is able to produce hard synthesis problem instances. The application of the test case instance generator for an evaluation of a novel DSE approach shows that the impact on performance is negligibly small compared to the solving complexity of the generated test instances.

1. Introduction

System-level Design Space Exploration (DSE) has been shown in many works (e.g., [1, 3, 6–8]) to be necessary in order to find good solutions for the often multi-objective optimization problem of embedded systems design. Such problems are defined at system level by a set of applications consisting of communicating tasks that have to be implemented onto hardware platforms with computational elements (CPUs, DSPs, etc.) and a communication infrastructure (routers, links, shared memory). A resulting implementation (resource allocation, tasks mapping, and scheduling) of an embedded system can be characterized by various quality characteristics (e.g., latency, throughput, energy requirement) that may be subject to optimization and/or have to fulfill several constraints. Thus, a DSE is imperative to find feasible solutions with Pareto-optimal properties.

As the system synthesis problem is NP-complete [1], an exhaustive search of all design points is impossible for reasonably large problems such that only a subset of all solutions can be obtained. Thus, the actual result of a DSE run is usually not the true Pareto-front but only an approximation thereof. In order to evaluate the performance of different runs or DSE approaches, quality indicator techniques must be used that measure the distribution of solutions in the approximation set.

Typically, a combination of convergence (distance to the true Pareto-front) and diversity (degree of distribution) are utilized to evaluate and compare the performances.

However, the specific problem instance in use is another important factor when evaluating a DSE technique as differently structured applications and assumptions can lead to both easier and harder optimization problems. On the one hand, in order to evaluate the performance of different DSE approaches with respect to each other, it is imperative that the DSE inputs are similar and easily reproducible. On the other hand, for the development of new techniques, test cases that represent a specific class of problems are desired to cover a large input space. That is, variant test cases with similar properties must be used to get meaningful results on the performance of a DSE technique for specific classes of problem instances.

In the work at hand, we propose a systematic test generation methodology for embedded systems design problems. It considers both the application and hardware architecture generation and is able to generate variant test cases with similar properties deterministically. As it is based on Answer Set Programming (ASP), it is easily expendable and platform independent.

The remainder of the paper is organized as follows: In Section 2, related work is discussed and an overview of DSE techniques for embedded systems design is given. Afterwards, we first specify the considered application and architecture model in Section 3 before we present our test case generator in Section 4. Finally, Section 5 concludes the paper.

2. Related Work

In the last decades, a number of approaches have been proposed to tackle the DSE in the area of embedded systems design. The works can be partitioned into (meta-)heuristics like evolutionary algorithms and particle swarm optimization (e.g., [3, 6, 13]), exact methods based on symbolic encoding techniques (e.g., [7, 8]) as well as hybrid approaches that combine symbolic encoding techniques with heuristic searches (e.g., [11]). Compared to purely heuristic search techniques, exact methods encode the DSE problem symbolically and systematically include all constraints into the solving process. Thus, only feasible solutions are found. However, such methods are only applicable to small problem instances. Meta-heuristics on the other hand, can also be utilized for large instances but may take a long time finding feasible solutions if design constraints are set tightly. As a remedy, hybrid methods encode the problem the symbolically and explore the search space of possible heuristics to find diverse solutions. This way, large problems can be considered while guaranteeing that only feasible solutions are found.

While the research on DSE has lead to variant approaches for the exploration itself, systematic methods to generate test instances are sparse. There exist two graph generation tools [2, 12] that have been proposed to be used as standard tools for the generation of random task graphs and synchronous data flow graphs (SDFGs), respectively. *Taskgraph For Free (TGFF)* [2] was especially designed to serve as a reference system for generating random task graphs for mapping and scheduling purposes. It constructs task graphs on the basis of two different algorithms, that can be configured with a number of parameters. While the first algorithm specifies the properties of the graph by the maximum *In/Out Degree* of nodes, the second algorithm generates series-parallel task graphs recursively with the option of additional edges between parallel units.

The authors of [12] proposed *SDF For Free (SDF^3)*, a tool to generate, analyze and visualize SDFGs. The generated graphs are connected, consistent, and deadlock free with user defined

characteristics. Furthermore, the tool consists of (user expendable) functions that are able to assign specific properties to actors and channels of the graph.

In contrast to the paper at hand, both SDF^3 and TGFF, however, only generate the application and do not consider the hardware architecture or mapping possibilities. While the task graph model considered in the paper at hand is similar to TGFF, SDF^3 is especially orientated towards the data flow model of computation that additionally allows the generation of cyclic applications. In following, we therefore present a holistic approach that is able to concurrently generate the application, the hardware architecture and assign mapping possibilities. As a result, complete problem instances are generated that can be used as input for any DSE methodology that can read text-based input files. Note that an ad-hoc random approach is not desirable as it may generate problem instances that are not solvable due to mapping (and scheduling) constraints. Furthermore, ad-hoc approaches have little to no influence on the solving complexity of generated test instances.

3. Prerequisites

In this Section, we formally describe the specification model including the application, the hardware architecture, and the mapping options that is produced by the generator and serves as input for the DSE. Furthermore, we lay out the basics of Answer Set Programming (ASP) which we utilize to implement the generator.

3.1. Specification Model

As depicted in Fig. 1, we model the system specification as a graph separated into a set of applications A, a heterogeneous architecture template P, and a set of mapping options M that connects the former two.

Application: The set of applications A consists of independent applications A_i that are specified in task-level granularity and modeled as a directed acyclic graph $A_i = (T, C, E)$. That is, each application contains sets of computational tasks T and communication messages C. A set of edges $E \subseteq T \times C \cup C \times T$ specifies dependency relations between the elements. Each message $c \in C$ has exactly one predecessor task, i.e. inter process communication is characterized in a point-to-point fashion. In order to model the individual complexity of tasks, the function $instructionCount : T \times IT \mapsto \mathbb{N}$ assigns to each task an integer number of instructions per instruction type $it \in IT$ (e.g., integer, floating point, memory, etc.). This property is utilized to determine the worst case execution time (WCET) for each mapping option.

Architecture Template: The architecture, or *platform* template $P = (V_P, V_R, L)$ consists of processing elements V_P and the communication infrastructure split into routers V_R and links L. The functions $staticPower : V_P \cup V_R \mapsto \mathbb{N}$ and $area : V_P \cup V_R \mapsto \mathbb{N}$ assign integer values to processing elements and routers such that they are annotated by individual area and static power requirements. Analogously, the functions $routingEnergy : L \mapsto \mathbb{N}$ and $routingDelay : L \mapsto \mathbb{N}$ assign energy and routing delay requirements to each link $l \in L$. That is, a message that is sent over a link l consumes $routingEnergy(l)$ energy and has a delay of $routingDelay(l)$[1]. Finally, for each processing element $p \in V_P$, the functions $cpi : V_P \times IT \mapsto \mathbb{N}$ and $epi : V_P \times IT \mapsto \mathbb{N}$ assign the required cycles

[1] In this paper, we assume messages to have a fixed size. Modeling larger messages is possible by introducing multiple messages between two tasks.

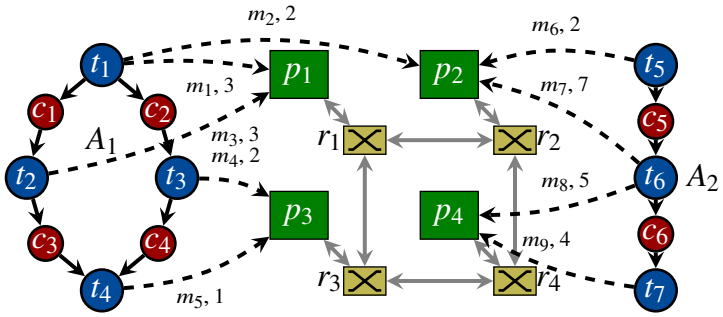


Figure 1: Specification example consisting of two applications A_1 and A_2, a 2×2 platform template with four processing elements p_{1-4} and routers r_{1-4}, and mapping options m_1 to m_9 annotated with WCET values.

and energy per processed instruction of type $it \in IT$. Note that bidirectional arrows represent two separate links. For example, the connection between p_1 and r_1 in Fig. 1 represents the directed edges $l_1 = (p_1, r_1)$ and $l_2 = (r_1, p_1)$.

Problem Instance: For each task, a set of mapping options $M \subseteq T \times V_P$ is specified. A mapping option $m = (t, p)$ indicates that task t may be executed on processing element p and is annotated with a WCET w as well as the dynamic energy e consumed by p when executing t. The WCET and dynamic energy of mapping option $m = (t, p)$ are calculated by Equations (1) and (2), respectively.

$$w(m) = \sum_{i \in IT} cpi(p, i) \cdot instructionCount(t, i) \tag{1}$$

$$e(m) = \sum_{i \in IT} epi(p, i) \cdot instructionCount(t, i) \tag{2}$$

Specifying several mapping options per tasks with different WCETs and energy annotations corresponds to the modeling of heterogeneous systems. Together with the applications and the platform template, the mapping options complete the problem instance $I = (A, P, M)$.

3.2. Answer Set Programming

To generate problem instances as introduced in the previous section, we implement the generator as Answer Set Programming (ASP) facts and rules. ASP is a programming paradigm that is tailored towards NP-hard search problems and is based on the stable model (*answer set*) semantics. Problems are formulated in a first-order input language as a set of facts and rules that are used to represent and infer domain knowledge, respectively. Facts, as the simplest constructs, are used to define a specific problem instance. Thus, they are unconditionally true and consist of only one atom (i.e., an n-ary predicate applied to n terms). Given the example in Fig. 1, the facts below encode the existence of task t_1 as well as its mapping options m_1 and m_2.

```
task(t1).  map(m1,t1,p1).  map(m2,t1,p2).
```
(3)

Rules are used to encode constraints and typically contain variables (i.e., names that start with uppercase letters) that are independent of a particular problem instance. For example, the following (choice) rule encodes the selection from mapping options.

```
{ bind(M,T,P) : map(M,T,P) } = 1 :- task(T).
```
(4)

Here, the ternary predicate `bind` (short `bind/3`) is inferred exactly once for every `task/1` predicate. Thus, an answer set containing both `bind(m1,t1,p1)` and `bind(m2,t1,p2)` simultaneously cannot exist.

Determining answer sets of logic programs (i.e., the combination of facts and rules) in ASP is a two-step process. First, the logic program is translated (*grounded*) into a variable free representation before it can be solved by an answer set solver that determines stable models (*solutions*). Accordingly, solving the example in Eqs. (3) and (4) produces two stable models (solutions): One with atom `bind(m1,t1,p1)` and the other with `bind(m2,t1,p2)`. A detailed description of the solving process is, however, out of scope of this paper and we refer to [5] for further information.

Note that, we do not use pure ASP but utilize some features of the answer set solver *clingo*5 [4] such as callback functions to generate IDs and implement a pseudo random number generator that enables platform independent results. These callback functions are evaluated during grounding and do not influence the determination of answer sets.

4. Generator

In this section, we present our methodology for generating system-level problem instances. The generator utilizes the constraint solving capabilities of ASP to define the desired characteristics of generated test cases. The most important properties of our methodology are listed below:

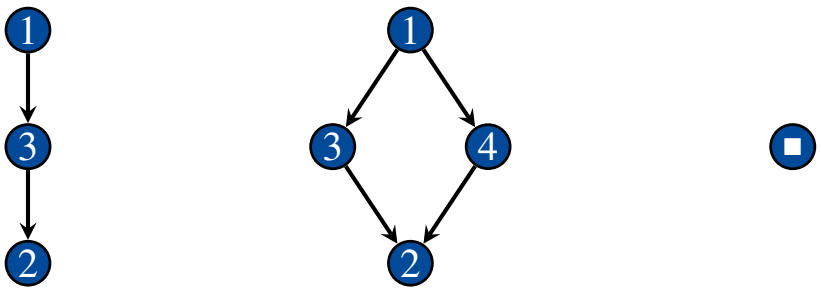
- **Systematic generation:** Rules encoded in ASP guarantee the compliance to desired characteristics of the resulting test case. This involves the number of tasks, messages, processing and communication elements as well as communication behavior of the application.
- **Varied test cases:** Utilizing the stable model semantics of ASP, the generator produces variant instances with similar characteristics as described above while the shape of the generated applications differ. This is important to evaluate DSE techniques with respect to various classes of applications.
- **Modularity:** The generation of application and architecture template are independent from each other, allowing the exchange of rules individually. Furthermore, as of the modular structure, additional domain-specific properties can be added easily.
- **Platform independence:** The generator is implemented in ASP allowing it to be executed on each system on which a compatible ASP solver is installed.

Next, we describe the implemented modules to generate applications, the hardware architecture, and mapping options before we propose a method on how to feasibly handle design constraint.

4.1. Application Module

For the application generator module, we consider the applications to be modeled as series-parallel graphs (SPGs). In this way, a wide range of application characteristics can be described by one versatile encoding. An SPG is a fully connected graph that consists of series and parallel patterns.[2] While a series pattern can be used to model direct dependencies between two tasks (or

[2]Note that without loss of generality, we only consider binary parallel patterns in this paper. This can be easily extended to parallel patterns with more than two branches.



(a) Series pattern **(b)** Parallel pattern **(c)** Terminal pattern

Figure 2: Different patterns considered in the series-parallel graph generation.

Listing 1: Application Encoding

```
1  series(P,A)   :- patterns(NS,NP,A), P=1..NS.
2  parallel(P,A) :- patterns(NS,NP,A), P=NS+1..NP+NS.
3  pattern(P,A)  :- series(P,A).
4  pattern(P,A)  :- parallel(P,A).
5  1 {contains(P1,P2,N,A) : pattern(P2,A), P1!=P2; contains(P1,term(N),N,A)} 1 :-
       ↪ parallel(P1,A), N=1..4.
6  1 {contains(P1,P2,N,A) : pattern(P2,A), P1!=P2; contains(P1,term(N),N,A)} 1 :-
       ↪ series(P1,A), N=1..3.
7  :- contains(P1,P2,X,A), contains(P1,P2,Y,A), X!=Y.
8  contains(P1,P2,A) :- contains(P1,P2,_,A).
9  :- contains(P1,P2,A), contains(P3,P2,A), pattern(P2,A), P1!=P3.
10 1{start(P,A) : pattern(P,A)}1 :- patterns(_,_,A).
11 reachable(P,A)  :- start(P,A).
12 reachable(P2,A) :- contains(P1,P2,A), reachable(P1,A).
13 :- not reachable(P,A), pattern(P,A).
14 contains_trans(P1, P2, A) :- contains(P1, P2, A), pattern(P2,A).
15 contains_trans(P1, P3, A) :- contains_trans(P1,P2,A), contains(P2, P3, A), pattern(P3, A).
16 :- contains_trans(P, P, A).
17 [...]
18 0 {instructions(task(TID,A),TYPE,N) : N=@getValue(instr_min,instr_max,seed,(TID,A,TYPE))} 1 :-
       ↪ task(TID,A), TYPE=1..instr_nr.
19 :- 0 #count {TYPE,(TID,A):instructions(task(TID,A),TYPE,_)} 0, task(TID,A).
20 instruction_exist(TYPE) :- instructions(_,TYPE,_).
21 :- not instruction_exist(TYPE), TYPE=1..instr_nr.
```

sub-applications) of an application, a parallel pattern models possible concurrent execution of two sub-applications. As an example, consider applications A_1 and A_2 in Figure 1. The former can be described as a parallel pattern where tasks t_2 and t_3 are able to be executed concurrently (and in arbitrary order). That is, they are only dependent on their common predecessor task t_1. The latter application, on the other hand, represents a series pattern where task t_5, t_6, and t_7 have to be executed in a strict order as they are form a dependency chain. Depending on the dominance of series or parallel patterns, the application allows for less or more concurrency, respectively.

An SPG is constructed recursively: Initially, an SPG has the form of a series (Fig. 2b), parallel (Fig. 2a) or terminal (Fig. 2c) pattern. While terminal nodes determine the stop criterion of the recursive construction process, each node of series and parallel patterns contains itself one of the patterns depicted in Fig. 2. Given a fixed number of series and parallel patterns to be generated, s and p respectively, this results in an overall number of $1 + 2 \cdot s + 3 \cdot p$ tasks and $2 \cdot s + 4 \cdot p$ edges.

The encoding of the series-parallel is depicted in Listing 1. At first, series and parallel patterns are inferred from the input `patterns/3` facts, defining the number of series and parallel patterns for application $A_i \in A$, viz. NS and NP. Line 3 and 4 define that both series and parallel are

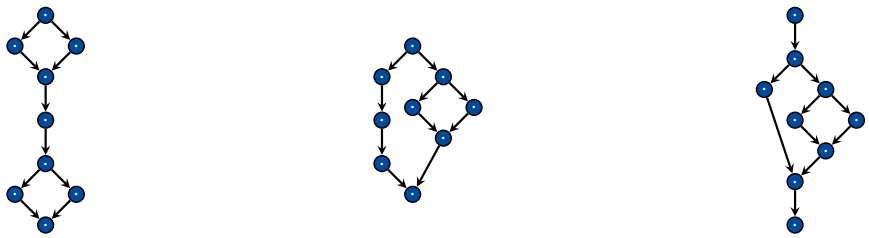


Figure 3: Three variant applications with similar characteristics. Each application contains one series and two parallel patterns, i.e., as input for the generator the atom `patterns(1,2,1)` was used.

Listing 2: Architecture Encoding

```
1  router(ID,X,Y,Z) :- ID=X+NX*(Y-1)+NX*NY*(Z-1), resources(NX,NY,NZ), X=1..NX, Y=1..NY, Z=1..NZ.
2  processor(ID,X,Y,Z) :- router(ID,X,Y,Z).
3  router(ID) :- router(ID,_,_,_).
4  processor(ID) :- processor(ID,_,_,_).
5  link(router(N),processor(N)) :- router(N,_,_,_), processor(N,_,_,_).
6  link(router(M),router(N)) :- router(M,X,Y,Z), router(N,X+1,Y,Z).
7  link(router(M),router(N)) :- router(M,X,Y,Z), router(N,X,Y+1,Z).
8  link(router(M),router(N)) :- router(M,X,Y,Z), router(N,X,Y,Z+1).
9  link(X,Y) :- link(Y,X).
10 0 { cpi(processor(RID),TYPE,N):N=@getValue(cpi_min,cpi_max,seed,(RID,TYPE)) } 1 :-
       ↪ processor(RID), TYPE=1..instr_nr.
11 epi(processor(R),TYPE,EPI) :- cpi(processor(R), TYPE, _),
       ↪ EPI = @getValue(epi_min,epi_max,seed,(R,TYPE)).
```

valid patterns. Afterwards, for each node of the parallel (line 5) and series (line 6) patterns, a child pattern is selected. That is, either another pattern or a terminal node is selected. To prevent invalid pattern constructs, lines 7-16 contain rules to prohibit several decisions. Line 7 provides that no pattern is contained at two different positions of another pattern. Lines 8 and 9 prohibit the situation that a pattern is contained in two different patterns at the same time, while lines 10-13 guarantee that the graph is fully connected for each application. Finally, lines 14-16 provide the transitive closure of the graph and make sure no cycle is present (a pattern must not contain itself transitively). An example of three applications with similar characteristics is depicted in Figure 3. Here, the input `patterns(1,2,1)` is used, specifying to generate an application containing one series pattern, two parallel patterns and having the ID "1". Note that for sake of brevity, the deduction of output predicates `task/2` and `comm/3` encoding tasks and messages for a specific application is not given in Listing 1. Lines 18 to 21 assign the number of instructions to each task. Furthermore, it is guaranteed that each tasks consists of at least one instruction type and that each instruction type is contained in at least on task. To this end, the 4-ary callback function `getValue(min,max,seed,ID)` returns a random integer value between *min* and *max* based on the *seed*. To reduce grounding overhead, this done only once for each *ID*. Note that all variables containing *min* and *max* are input parameters.

4.2. Architecture Module

The hardware architecture module generates a 3-dimensional grid of routers that are each connected to a processing element via two independent communication links. An example of a resulting platform is given in Figure 1. Here, the routers $r_1 - r_4$ form a grid of size $2 \times 2 \times 1$ and are connected to four processing elements $p_1 - p_4$. The encoding is shown in Listing 2. At first, the number and

Listing 3: Mapping Encoding

```
1 N{map(ID,task(T,A),processor(R)):processor(R),ID=@getId(m,(T,A,R))}N:-task(T,A),
      ↪ N=@getValue(map_min, map_max, seed,(T,A)).
2 :- map(ID, T, R), instructions(T,TYPE,_), not cpi(R,TYPE,_).
3 executionTime(MID,TIME) :- map(MID, T, R), TIME=#sum{CYCLES,TYPE:instructions(T,TYPE,INS),
      ↪ cpi(R,TYPE,IPC), CYCLES=INS*CPI}.
4 dynamicEnergy(MID,E) :- map(MID, T, R), E=#sum{ENERGY,TYPE:instructions(T,TYPE,INS),
      ↪ epi(R,TYPE,EPI), ENERGY=INS*EPI}.
```

IDs of routers are determined by interpretation of the input atom `resources/3` that represents the grid size in each dimension X, Y, and Z. For each router, an accompanying processing element (`processor/4`) is inferred in line 2 before both 4-ary predicates `router/4` and `processor/4` are projected to unary predicates in lines 3 and 4. Lines 5 to 9 define the communication links (`link/2`) between routers and processing elements. At first, a link is created between a router and its processing element having the same ID. Afterwards, in lines 6 to 8, a link between each two neighboring routers is added. Finally, the last link rule assures bidirectional links between elements. The last three lines generate random *CPI* and *EPI* values for each processing element and instruction type. Note that a processor may not support a specific instruction type.

4.3. Mapping Module

Given the application and architecture module, for the completion of a problem instance $I = (\mathcal{A}, P, M)$, mapping options as well as further properties (energy and timing requirements) have to be determined. To this end, in the mapping module, a random number of mapping options is generated for each tasks that has been created by the application module. Afterwards, energy and timing requirements are determined based on instruction count, instruction type, as well as *CPI* and *EPI* values generated in other modules.

The encoding is given in Listing 3. Line 1 generates a random number of mapping options. Here, the binary callback function `getID(BASE,ARG)` determines a unique identifier for the mapping option based on the identifiers of the task, the resource it is mapped to and the application number. Line 2 guarantees that no mapping is inferred, that binds a task to an incompatible processing element, i.e., if the processing element does not support a specific instruction type, no task containing such instructions must be mapped to that resource. Finally, the execution time and dynamic energy for a specific mapping option are calculated in lines 3 and 4, respectively.

4.4. Constraint Generation

Generally, in order to evaluate DSE approaches, it becomes imperative to provide a large variety of test cases. To this end, the generator is used to generate different problem classes with varying characteristics. That is, each problem instance that belongs to a specific class has similar characteristics including the number of series and parallel patterns, the size of the architecture, and the range of property values. Utilizing each of the generated problem instances as input, the design space exploration is carried out and can be evaluated with respect to varying problem instances.

In addition to the properties discussed before, real system implementations have to fulfill several constraints, like maximum power dissipation and latency/throughput requirements. One possible approach to deal with such demands in the generator is to utilize random numbers. However,

a random constraint generation does not work properly as the problem instance would either be over-constrained (i.e., there are no feasible solutions to be found) or under-constrained (i.e., every solution found during DSE is feasible). Unfortunately, as the system synthesis problem is NP-complete, non-trivial upper bounds (and lower bounds) of constraints are hard to calculate.

Therefore, we propose a two-step approach based on the work of [9] utilizing the ASP modulo Theories (ASPmT) paradigm to tightly integrate non-linear constraints analyzing techniques into the ASP solving. ASPmT is a paradigm to integrate background theories (like difference logic) into the ASP solver that enables the consideration of non-linear constraints. This way, after a problem instance is determined, we are able to analyze valid solutions in the background theory. Finally, this information is used to calculate hard constraints for a given problem instance by weighting the analysis results with a user specified complexity factor. This way, instances of the same class can be set to a similar relative difficulty. This approach was utilized to generate the test instances of [10]. Here, the generation of test case instances (including constraint generation) has been executed in a matter of a few seconds while the DSE ran for 30 minutes per instance. Compared to the solving complexity of the generated instances, the generator itself only has a small impact on performance.

5. Conclusion

In this paper, we have presented a novel methodology for generating holistic problem instances to evaluate Design Space Exploration (DSE) techniques. Based on Answer Set Programming (ASP), our approach provides a modular generation of different parts of the synthesis problem instance. In order to allow for systematic generation of applications and hard architectures, we utilize the class of series-parallel task graphs that are characterized by their contained number of series and parallel patterns as well as connected grids to model the communication infrastructure. This way, only a few input parameters have to be used to create a number of different problem instances with similar characteristics. Random *CPI* and *EPI* values for processing elements in combination with different instruction types are utilized to provide realistic heterogeneous problem instances where tasks with different instruction combinations behave deterministically when mapped to various processing elements. Finally, in order to generate test case instances with user defined solving complexity regarding desired design constraints, we propose to utilize the ASPmT paradigm.

Acknowledgment

This work was funded by the German Science Foundation (DFG) under grants HA 4463/4-1 and SCHA 550/11-1.

References

[1] Blickle, T., J. Teich, and L. Thiele: *System-level synthesis using evolutionary algorithms*. Design Automation for Embedded Systems, 58:23–58, 1998. `http://link.springer.com/article/10.1023/A:1008899229802`.

[2] Dick, R. P., D. L. Rhodes, and W. Wolf: *Tgff: task graphs for free*. In *Proceedings of the Sixth International Workshop on Hardware/Software Codesign, 1998. (CODES/CASHE '98)*, pages 97–101, Mar 1998. `http://ziyang.eecs.umich.edu/~dickrp/tgff/`.

[3] Ferrandi, F., P. L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo: *Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 29(6):911–924, 2010, ISSN 0278-0070.

[4] Gebser, M., R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and P. Wanko: *Theory solving made easy with clingo 5*. In *Technical Communications of the Thirty-second International Conference on Logic Programming (ICLP'16)*, 2016.

[5] Gebser, M., R. Kaminski, B. Kaufmann, J. Romero, and T. Schaub: *Progress in clasp series 3*. In *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, pages 368–383, 2015.

[6] Jia, Z.J., A. Núñez, T. Bautista, and A.D. Pimentel: *A two-phase design space exploration strategy for system-level real-time application mapping onto mpsoc*. Microprocessors and Microsystems, 38(1):9 – 21, 2014, ISSN 0141-9331. `http://www.sciencedirect.com/science/article/pii/S0141933113001361`.

[7] Khalilzad, N., K. Rosvall, and I. Sander: *A modular design space exploration framework for multiprocessor real-time systems*. In *Proceedings of 2016 Forum on Specification and Design Languages (FDL)*, pages 1–7, 2016.

[8] Lukasiewycz, M., M. Glass, C. Haubelt, and J. Teich: *Efficient symbolic multi-objective design space exploration*. In *Proceedings of 2008 Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 691–696, 2008.

[9] Neubauer, K., P. Wanko, T. Schaub, and C. Haubelt: *Enhancing symbolic system synthesis through ASPmT and partial assignment evaluation*. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 306–309, Lausanne, 2017.

[10] Neubauer, K., P. Wanko, T. Schaub, and C. Haubelt: *Exact multi-objective design space exploration using aspmt*. In *Proceedings of Design, Automation and Test in Europe (DATE)*, Dresden, 2018.

[11] Schlichter, T., M. Lukasiewycz, C. Haubelt, and J. Teich: *Improving system level design space exploration by incorporating SAT-solvers into multi-objective evolutionary algorithms*. In *Proceedings of IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, 2006, ISBN 0-7695-2533-4.

[12] Stuijk, S., M.C.W. Geilen, and T. Basten: *SDF^3: SDF For Free*. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, pages 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006. `http://www.es.ele.tue.nl/sdf3`.

[13] Thompson, M. and A. D. Pimentel: *Exploiting domain knowledge in system-level mpsoc design space exploration*. Journal of Systems Architecture, 59(7):351 – 360, 2013, ISSN 1383-7621. `http://www.sciencedirect.com/science/article/pii/S1383762113001045`.