Symbolic System Synthesis Using Answer Set Programming

Benjamin Andres[1], Martin Gebser[1], Torsten Schaub[1], Christian Haubelt[2], Felix Reimann[3], and Michael Glaß[3]

[1] Institute for Computer Science,
University of Potsdam, Germany
{bandres,gebser,torsten}@cs.uni-potsdam.de
[2] Institute of Applied Microelectronics and Computer Engineering
University of Rostock, Germany
christian.haubelt@uni-rostock.de
[3] Chair for Hardware/Software Co-Design
University of Erlangen-Nuremberg, Germany
{felix.reimann,glass}@cs.fau.de

Abstract. Recently, Boolean Satisfiability (SAT) solving has been proposed to tackle the increasing complexity in high-level system design. Working well for system specifications with a limited amount of routing options, they tend to fail for densely connected computing platforms. This paper proposes an automated system design approach employing Answer Set Programming (ASP). ASP provides a stringent semantics, allowing for an efficient representation of routing options. An automotive case-study illustrates that the proposed ASP-based system design approach is competitive for sparsely connected computing platforms, while it outperforms SAT-based approaches for dense Networks-on-Chip by an order of magnitude.

1 Introduction

Embedded computing systems surround us in our daily life. They are application-specific computing systems embedded into a technical context. Examples of embedded computing systems are automotive, train, and avionic control systems, smart phones, medical devices, home and industrial automation systems, etc. In contrast to general purpose computing systems, embedded computing systems are not only optimized for performance; they additionally have to satisfy power, area, reliability, real-time constraints, to name just a few. As a consequence, the computing platform is adapted to the given application. At the system-level, however, resulting embedded computing platforms are still as complex as heterogeneous multi-processor systems, i.e., several different processing cores are interconnected and the memory subsystem is optimized for the application as well. Finally, the application has to be mapped optimally onto the resulting computing platform. In summary, embedded computing system design includes many interdependent design decisions.

The increasing complexity of interdependent decisions in embedded computing systems design demands for compact design space representations and highly efficient

automatic decision engines, resulting in automatic *system synthesis* approaches. Especially, formal methods have shown to be useful in past. (Pseudo-)Boolean Satisfiability (PB/SAT; [2]) solving has been successfully applied in the past to such problems. In particular, explicit modeling of routing decisions in PB formulas has recently enhanced the range of applicability of PB/SAT solvers in synthesizing networked embedded systems [10].

PB/SAT-based approaches to system synthesis work well in the presence of system specifications offering a limited amount of routing options. Such system specifications can be found, e.g., in the automotive or bus-based Multi-Processor System-on-Chip (MPSoC) domain. However, there is a trend towards densely connected networks also in the embedded systems domain. In fact, future MPSoCs are expected to be composed of several hundred processors connected by Networks-on-Chip (NoC) [4]. Hence, system synthesis approaches will face vast design spaces for densely connected networks, resulting in prohibitively long solving times when using PB/SAT-based approaches.

In this paper, we investigate system synthesis scenarios relying on reachability for message routing. We propose a formal approach employing Answer Set Programming (ASP; [1]), a solving paradigm stemming from the area of Knowledge Representation. In contrast to PB/SAT, ASP provides a rich modeling language as well as a more stringent semantics, which allows for succinct design space representations. In particular, ASP supports expressing reachability directly in the modeling language. As a result, much smaller problem descriptions lead to significant reductions in solving time for densely connected networks.

In what follows, we assume some familiarity with ASP, its semantics as well as its basic language constructs. A comprehensive treatment of ASP can be found in [1, 7]. Our encodings are written in the input language of *gringo* 3 [6].

After surveying related work, Section 3 introduces the system synthesis setting studied in the sequel. Section 4 provides dedicated ASP formulations of system synthesis. The experiments in Section 5 illustrate the effectiveness of our ASP-based approach. Section 6 concludes the paper.

2 Related Work

Symbolic system synthesis approaches based on Integer Linear Programming (ILP) can be found in the area of hardware/software partitioning (cf. [13]). Such approaches were often limited to the classical bipartitioning problem, i.e., the target platform is composed of a CPU and an FPGA. An extension towards multiple resources and a simple single-hop communication mapping can be found in [3]. In the same work, SAT is reduced to the problem of computing feasible allocations and bindings in platform-based system synthesis approaches, thereby showing that system synthesis is NP-complete. In turn, [8] shows how to reduce the system synthesis problem to SAT in polynomial time; this allows for symbolic SAT-based system synthesis. An analogous approach based on binary decision diagrams is presented in [12]. Since the space requirements of binary decision diagrams may grow exponentially, it could only be applied to small systems. In [11], a first approach to integrate linear constraint checking into SAT-based system synthesis is reported, leading to a PB problem encoding. All aforementioned approaches

still use simple single-hop communication as underlying model. However, single-hop communication models are no longer appropriate when designing complex multi-core systems.

In [10], the authors show how to perform symbolic system synthesis including multi-hop communication routing with PB solving techniques. This work is closely related to ours, and we use it as a starting point for the work at hand. We show that the PB-based approach published in [10] does not scale well for system specifications permitting many routing options. By reformulating the PB representation in ASP and exploiting semantic features in expressing reachability, symbolic system synthesis can be applied to more complex system specifications based on densely connected communication networks.

The potential of ASP for system synthesis was already discovered in [9], where it was shown to outperform an ILP-based approach by several orders of magnitude. In contrast to our work, the system synthesis problem considered in [9] does not involve multi-hop communication routing. Moreover, contrary to the genuine ASP encoding(s) developed in Section 4, the one in [9] was derived from an ILP specification without making use of any elaborate ASP features.

3 Symbolic System Synthesis

System synthesis comprises several design phases: the allocation of a computing platform, the binding of tasks onto allocated resources, and the scheduling of tasks for resolving resource conflicts. Each phase, viz., allocation, binding, and scheduling, can be performed either statically or dynamically. We here assume that allocation and binding are accomplished statically, whereas scheduling is realized dynamically. Accordingly, we concentrate on allocation and binding in the sequel.

In order to automate the synthesis of a system implementing an application, the application is modeled by a *task graph* (T, E_T). Its vertices T represent *tasks* and are bipartitioned into *process* tasks P and *communication* tasks C, that is, $T = P \cup C$ and $P \cap C = \emptyset$. The directed edges $E_T \subseteq (P \times C) \cup (C \times P)$ model data and control dependencies between tasks, where every communication task has exactly one predecessor and an arbitrary (positive) number of successors, thus assuming single-source multicast communication.

An exemplary task graph is shown in the upper part of Figure 1. The leftmost task p_s reads data from a sensor and sends it to a master task p_m via communication task c_1. The master task then schedules the workload and passes data via communication task c_2 on to the worker tasks p_1 and p_2. Both workers send their results back to the master via communication tasks c_3 and c_4. Finally, the master uses the combined result to control an actuator task p_a via communication task c_5.

An architecture template, representing all possible instances of a computing platform, is modeled by a *platform graph* (R, E_R). Its vertices R represent resources like processors, buses, memories, etc., and the directed edges $E_R \subseteq R \times R$ model communication connections between them. The lower part of Figure 1 shows a platform graph containing six computational and two communication resources along with 18 connections. Any subgraph of a platform graph constitutes a computing platform instance.

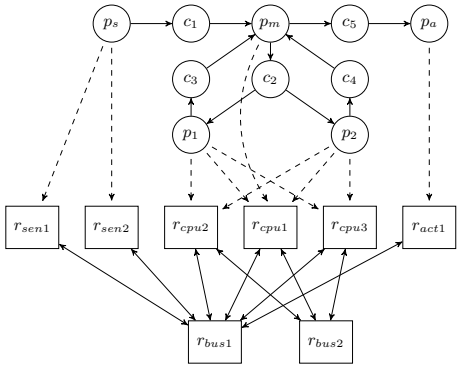

Fig. 1: A system model consisting of a task graph, a platform graph and mapping options.

Given a task graph (T, E_T) and a platform graph (R, E_R), mapping options of tasks $t \in T$ are determined by $R_t \subseteq R$, providing resources on which t can be implemented. Assuming that communication tasks can be routed via every resource, the mapping options of process tasks are indicated by dashed arrows in Figure 1, while the (unrestricted) options of communication tasks are not displayed explicitly.

Following [10], the system synthesis problem can be defined as follows. For (T, E_T) and (R, E_R) as above, select an allocation $A \subseteq R$ of resources and a binding $b : T \to 2^R$ such that the following conditions are fulfilled:

- $b(t) \subseteq R_t$ for each task $t \in T$,
- $|b(p)| = 1$ for each process task $p \in P$, and
- for each $(p, c) \in (P \times C) \cap E_T$, there is an arborescence $(b(c), E)$ with root $r \in b(p)$ such that $E \subseteq E_R$ and $\{\hat{r} \mid (c, \hat{p}) \in E_T, \hat{r} \in b(\hat{p})\} \subseteq b(c)$.

These conditions require that each process task is mapped to exactly one resource and that each communication task can be routed (acyclicly) from the sender to resources of its targets.

Figure 2 shows a feasible implementation for the example in Figure 1, consisting of the resource allocation $A = \{r_{sen1}, r_{bus1}, r_{bus2}, r_{cpu1}, r_{cpu2}, r_{cpu3}, r_{act1}\}$ along with the following mapping b:

$$\begin{array}{ll} b(p_s) = \{r_{sen1}\} & b(c_1) = \{r_{sen1}, r_{bus1}, r_{cpu1}\} \\ b(p_m) = \{r_{cpu1}\} & b(c_2) = \{r_{cpu1}, r_{bus1}, r_{cpu2}, r_{bus2}, r_{cpu3}\} \\ b(p_1) = \{r_{cpu2}\} & b(c_3) = \{r_{cpu2}, r_{bus2}, r_{cpu1}\} \\ b(p_2) = \{r_{cpu3}\} & b(c_4) = \{r_{cpu3}, r_{bus2}, r_{cpu1}\} \\ b(p_a) = \{r_{act1}\} & b(c_5) = \{r_{cpu1}, r_{bus1}, r_{act1}\} \end{array}$$

For clarity communication task mappings are omitted in Figure 2. Instead, the routing of c_2 is shown. Leading from the resource r_{cpu1} of the master task p_m over r_{bus1}, r_{cpu2}, and r_{bus2} to r_{cpu3}, thus visiting the resources r_{cpu2} and r_{cpu3} of the workers p_1 and p_2.

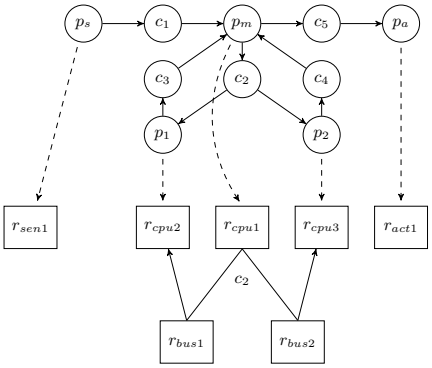

Fig. 2: A feasible implementation of the example system model. Only the routing of c_2 is shown.

PB/SAT-based approaches express system synthesis in terms of (Pseudo-)Boolean formulas. In particular, the PB encoding in [10] relies on the following kinds of Boolean variables:

- a variable $\mathbf{r}$ for each resource $r \in R$, indicating whether r is allocated ($\mathbf{r} = 1$) or not ($\mathbf{r} = 0$),
- a variable $\mathbf{t}_r$ for each task $t \in T$ and each of its mapping options $r \in R_t$, indicating whether t is bound onto r, and
- variables $\mathbf{c}_{r,i}$ for each communication task $c \in C$, its routing options $r \in R_c$, and $i \in \{0, \ldots, n\}$ for some integer n, indicating whether c is routed over r at step i.

The following constraints on such variables were used in [10]:

$$\textstyle\sum_{r \in R_p} \mathbf{p}_r = 1, \qquad \forall p \in P \tag{A}$$

$$\textstyle\sum_{r \in R_c} \mathbf{c}_{r,0} = 1, \qquad \forall c \in C \tag{B}$$

$$\mathbf{p}_r - \mathbf{c}_{r,0} = 0, \qquad \forall c \in C, p \in \{\hat{p} \mid (\hat{p}, c) \in E_T\}, r \in R_p \cap R_c \tag{C}$$

$$\mathbf{c}_r - \mathbf{p}_r \geq 0, \qquad \forall p \in P, c \in \{\hat{c} \mid (\hat{c}, p) \in E_T\}, r \in R_p \cap R_c \tag{D}$$

$$\textstyle\sum_{i=0}^{n} \mathbf{c}_{r,i} \leq 1, \qquad \forall c \in C, r \in R_c \tag{E}$$

$$\textstyle\sum_{i=0}^{n} \mathbf{c}_{r,i} - \mathbf{c}_r \geq 0, \qquad \forall c \in C, r \in R_c \tag{F}$$

$$\mathbf{c}_r - \mathbf{c}_{r,i} \geq 0, \qquad \forall c \in C, r \in R_c, i \in \{0, \ldots, n\} \tag{G}$$

$$-\mathbf{c}_{r,i} + \textstyle\sum_{\hat{r} \in R_c, (\hat{r}, r) \in E_R} \mathbf{c}_{\hat{r}, i-1} \geq 0, \qquad \forall c \in C, r \in R_c, i \in \{1, \ldots, n\} \tag{H}$$

$$\mathbf{r} - \mathbf{p}_r \geq 0, \qquad \forall p \in P, r \in R_p \tag{I}$$

$$\mathbf{r} - \mathbf{c}_r \geq 0, \qquad \forall c \in C, r \in R_c \tag{J}$$

$$-\mathbf{r} + \textstyle\sum_{p \in P, r \in R_p} \mathbf{p}_r + \sum_{c \in C, r \in R_c} \mathbf{c}_r \geq 0, \qquad \forall r \in R \tag{K}$$

In words, (A) requires each process task to be mapped to exactly one resource. Jointly, (B) and (C) imply that each communication task has exactly one root matching the resource of its sending task. In addition, (D) makes sure that the resources of all targets

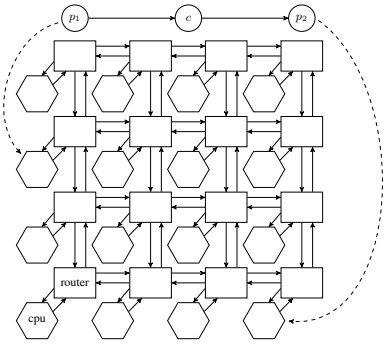

Fig. 3: A possible mapping of two communicating processes to resources connected via a 4x4 mesh network.

are among those of a communication task. For excluding cyclic routing, (E) asserts that the step at which a resource is visited upon performing a communication task is unique. By means of (F) and (G), the resources visited at particular steps (i.e., $\mathbf{c}_{r,i} = 1$ for some $i \in \{0, \ldots, n\}$) are synchronized with the ones assigned (i.e., $\mathbf{c}_r = 1$) to a communication task. The requirement that resources visited at successive steps must be connected in the underlying platform graph is expressed by (H). Finally, (I), (J), and (K) extract allocated resources r, indicated by $\mathbf{r} = 1$, from process and communication tasks such that $\mathbf{p}_r = 1$ or $\mathbf{c}_r = 1$, respectively.

As detailed in [10], a Boolean variable assignment satisfying (A)–(K) provides a feasible implementation via resources r such that $\mathbf{r} = 1$, where each process task p is bound onto the resource r given by $\mathbf{p}_r = 1$ and communication tasks c are routed via resources r according to steps i such that $\mathbf{c}_{r,i} = 1$.

The described approach to system synthesis works well for sparsely connected networks, inducing a limited amount of routing options. However, the representation of routing options, governed by (H), scales proportionally to $|E_R| * |R|$, given that resources may be pairwisely connected and each resource may be visited in the worst case. As a consequence, for densely connected networks, the size required for a step-based representation of routing options can be prohibitively large. For example, let us consider possible routes from (the resource of) a sender p_1 to p_2 available in the 4x4 mesh network shown in Figure 3. The longest of these routes passes all 16 routers and potentially visit any of them at each of the 15 intermediate steps. This yields $16*15 = 240$ instances of (H) per communication task to represent the message exchange between routers. On the other hand, for inductively verifying whether a message reaches its target(s), it is sufficient to consider individual routing hops without relying on an explicit order given by steps. The latter strategy scales linearly to $|E_R|$, thus avoiding a significant blow-up in space. As the semantics of ASP inherently supports efficient representations of inductive concepts like reachability, the potential space savings motivate our desire to switch from the PB-based approach in [10] to using ASP instead.

4 ASP-based System Synthesis

As common in ASP, we represent the system synthesis problem by facts describing a problem instance along with a generic encoding. To this end, we define the ASP instance for a task graph $(P \cup C, E_T)$ and a platform graph (R, E_R) along with the underlying mapping and routing options, $(R_p)_{p \in P}$ and $(R_c)_{c \in C}$, as follows:

$$\begin{aligned}
\{\mathtt{pt(p)}. \mid \mathtt{p} \in P\} \ \cup \\
\{\mathtt{send(p,c)}. \mid (\mathtt{p},\mathtt{c}) \in E_T, \mathtt{p} \in P, \mathtt{c} \in C\} \ \cup \\
\{\mathtt{read(p,c)}. \mid (\mathtt{c},\mathtt{p}) \in E_T, \mathtt{p} \in P, \mathtt{c} \in C\} \ \cup \\
\{\mathtt{pr(p,r)}. \mid \mathtt{p} \in P, \mathtt{r} \in R_\mathtt{p}\} \ \cup \\
\{\mathtt{cr(c,r)}. \mid \mathtt{c} \in C, \mathtt{r} \in R_\mathtt{c}\} \ \cup \\
\{\mathtt{edge(r,s)}. \mid (\mathtt{r},\mathtt{s}) \in E_R\} \ \cup \\
\{\mathtt{s(i)}. \mid \mathtt{i} \in \{1,\ldots,n\}\}
\end{aligned} \tag{1}$$

While the first six sets capture primary constituents of a problem instance, the introduction of atoms $\mathtt{s(i)}$ for $1 \leq \mathtt{i} \leq n$ is needed to account for the PB formulation in [10] in a faithful way.

Two alternative ASP encodings of system synthesis are shown in Figure 4(a) and 4(b). Essentially, they reformulate the constraints (A)–(K) from Section 3 in the input language of ASP to make sure that every answer set corresponds to a feasible system implementation. To this end, the rule in Line 2 of each encoding specifies that every processing task provided in an instance must be mapped to exactly one of its associated options. Observe that the mapping of processing tasks `p` to resources `r` is represented by atoms $\mathtt{map(p,r)}$ in an answer set. This provides the basis for further specifying communication routings.

Despite of syntactic differences, the step-oriented encoding ASP(S) in Figure 4(a) stays close to the original PB formulation of constraints, given in (A)–(K). In particular, it uses atoms $\mathtt{reached(c,r,i)}$ to express that some message of communication task `c` is routed over resource `r` at step `i`. Note that the omission of lower and upper bounds for the cardinality constraint in the rule form Line 8 means that there is no restriction on the number of atoms constructed by applying the rule. The (trivially satisfied) cardinality constraint is still important because, it allow us to successively construct $\mathtt{reached(c,r,i)}$. Given such atoms, instantiations of the rule in Line 12 (where "_" stands for an unreused anonymous variable) further provide us with projections $\mathtt{reached(c,r)}$. These are used in the integrity constraints in Line 14 and 16, excluding cases where a communication task is routed over the same resource at more than one step or does not reach some of its targets, respectively. Finally, projections via the rules in Line 19 and 20 provide the collection of resources allocated in an admissible system layout, similar to the (redundant) variables `r` in (I)–(K).

While the step-oriented encoding ASP(S) aims at being close to the constraints in (A)–(K), the encoding in Figure 4(b), denoted by ASP(R), utilizes ASP's "built-in" support of recursion to implement routing without step counting. To still guarantee an acyclic routing of communication tasks, the idea of ASP(R) is to (recursively) construct non-branching routes from resources of communication targets back to the resource of a sending task, where the construction stops. This recursive approach connects each encountered target resource to exactly one predecessor, where the only exception is due to

```
1  % map each process task to a resource                    (A)
2  1 { map(P,R) : pr(P,R) } 1 :- pt(P).

4  % step zero of communication task                      (B,C)
5  reached(C,R,0) :- send(P,C), map(P,R), cr(C,R).
6  % forward steps of communication task                    (H)
7  { reached(C,S,I+1) : cr(C,S) : edge(R,S) }
8                :- reached(C,R,I), s(I+1).

10 % resources of communication task                      (F,G)
11 reached(C,R) :- reached(C,R,_).
12 % reach each resource at most once                      (E)
13  :- reached(C,R), 2 { reached(C,R,_) }.
14 % reach communication target resources                  (D)
15  :- read(P,C), map(P,R), not reached(C,R).

17 % allocated resources                                (I,J,K)
18 allocated(R) :- map(_,R).
19 allocated(R) :- reached(_,R).
```

(a) Step-oriented encoding ASP(S).

```
1  % map each process task to a resource                    (A)
2  1 { map(P,R) : pr(P,R) } 1 :- pt(P).

4  % root resource of communication task                  (B,C)
5  root(C,R) :- send(P,C), map(P,R).
6  % resources of communication task per target
7  sink(C,R,P) :- read(P,C), map(P,R), cr(C,R).
8  sink(C,R,P) :- sink(C,S,P), reached(C,R,S).
9  % reach communication root resource                      (D)
10  :- read(P,C), root(C,R), not sink(C,R,P).

12 % resources of communication task                      (F,G)
13 reached(C,R) :- sink(C,R,_).
14 % backward hops of communication task                  (E,H)
15 1 { reached(C,R,S) : cr(C,R) : edge(R,S) } 1
16           :- reached(C,S), not root(C,S).

18 % allocated resources                                (I,J,K)
19 allocated(R) :- map(_,R).
20 allocated(R) :- reached(_,R).
```

(b) Recursive encoding ASP(R).

Fig. 4: Two alternative ASP encodings of system synthesis.

the sender of a communication task, whose resource, specified by an atom $\texttt{root}(\texttt{c},\texttt{r})$, is not connected back. Finally, the integrity constraint in Line 10 requires that each target of a communication task is located on a route starting at the sender's resource. Note that the target-driven routing approach implemented in ASP(R) intrinsically omits redundant message hops (not leading to communication targets). The same strategy could also be applied in step counting by modifying the constraints in (A)–(K) as well as our previous encoding ASP(S) accordingly. In view of this, the varied encoding idea is not the real achievement of ASP(R), while abolishing one problem dimension by disusing explicit step counters is.

5 Experiments

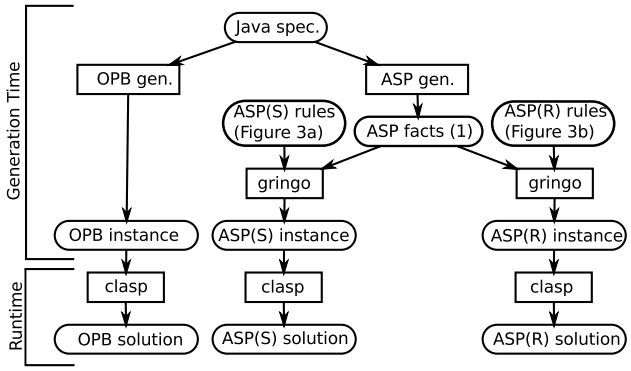

Fig. 5: Workflows of symbolic system synthesis approaches.

For evaluating our approach, we conducted systematic experiments contrasting our two ASP encodings, ASP(S) and ASP(R), in terms of design space representation size and solving time. In addition, we compare our methods to the original (sophisticated) PB-based synthesis tool from [10], which like ASP(S) uses steps to express routing. To this end, we consider both a real-world example consisting of a sparsely connected industrial system model as well as series of crafted mesh network system models of varying sizes.

The real-world example models an automotive subsystem including four applications of different criticality and characteristic, amongst others a multimedia/infotainment control and brake-by-wire. Overall, the applications involve 45 process tasks, communicating via 41 messages. The target platform offers 15 Electronic Control Units (ECUs), 9 sensors, and 5 actuators to execute the process tasks. For communication, up to three field buses (CAN or FlexRay), connected by a central gateway, are available. In addition, sensors and actuators are connected to ECUs via LIN buses. The case study, in particular when applying further design constraints, e.g., regarding bus load, can be viewed as a complex specification that tends to max out common synthesis approaches solely based on (greedy) heuristics. However, the PB-based approach solves

this problem efficiently, particularly due to the communication topology including a central gateway, resulting in a modest amount of routing options.

We ran the real-world example with the three approaches illustrated in Figure 5, all of which start from a common Java class specifying a system model (like the one shown in Figure 1). With the PB-based approach, the Java specification is directly converted into a PB instance (in OPB format) by the PB generator used also in [10]. Unlike this, with our two ASP-based approaches, the generation of facts describing a problem instance merely requires a syntactic conversion from the Java specification to the format in (1), from where the ASP grounder *gringo* (version 3.0.3) instantiates either of our encodings, ASP(S) or ASP(R), wrt the generated facts. With all three approaches, the generation phase results in standardized text formats, processable by the combined PB and ASP solver *clasp* (version 2.0.3; [5]). Let us note that ASP instance generation, including the conversion to facts and instantiation, runs quickly (only a few seconds on the largest of our benchmarks); on the other hand, PB instance generation can take significant time (up to five hours on the largest benchmarks we tried), which is because the PB generator performs nontrivial simplifications and, in contrast to ASP grounders, is not optimized towards low-level performance. After instance generation, accomplished offline, we measured (sequential) runtimes of *clasp* on a Linux machine equipped with 3.4GHz Intel Xeon CPUs and 32GB RAM. The search strategies of *clasp* were configured via command-line switches `--heuristic=vsids` and `--save-progress`, which in preliminary experiments turned out to be helpful for solving both PB and ASP instances. Then, the real-world example could be solved by *clasp* in less than a second for all three instance kinds, PB, ASP(S), and ASP(R). As mentioned above, this can be explained by the centralized communication topology in the example, so that routing options are rather limited.

In order to compare the three approaches also on densely connected networks, we generated series of synthesis problems wrt mesh network structures, scaling mesh size and number of process tasks. In these problems, each task can be bound onto a number of processors proportional to mesh size and communicates to one other process task; task mapping options and communication targets were selected randomly. In order to compensate for randomness in problem generation, we report averages over 16 distinct instances per mesh size and task number. Also note that all generated instances are satisfiable. In view of longer runtimes than before, we restricted single runs of *clasp* on a PB, ASP(S), or ASP(R) instance to 300 seconds time. Noise effects are excluded by taking the mean runtime over three (reproducible) runs of *clasp* per instance.

Figure 6 displays average numbers of constraints, as reported by *clasp*, and average runtimes of *clasp*, with timeouts taken as 300 seconds, over mesh networks of quadratic sizes (2x2, 3x3, ...) and increasing task numbers (10, 20, ...), both given along the x-axes; standard deviations are shown as vertical bars through measurements. The average numbers of constraints reported in the left chart provide an indication of problem representation size incurred by PB, ASP(S), and ASP(R). We observe regular scalings here, and ASP(S) is clearly the most space-consuming approach. In fact, the direct PB representation saves about half of the constraints of ASP(S) by virtue of the PB generator's simplifications. However, for larger mesh sizes, the recursive formulation of reachability in ASP(R) yields much more succinct problem representations

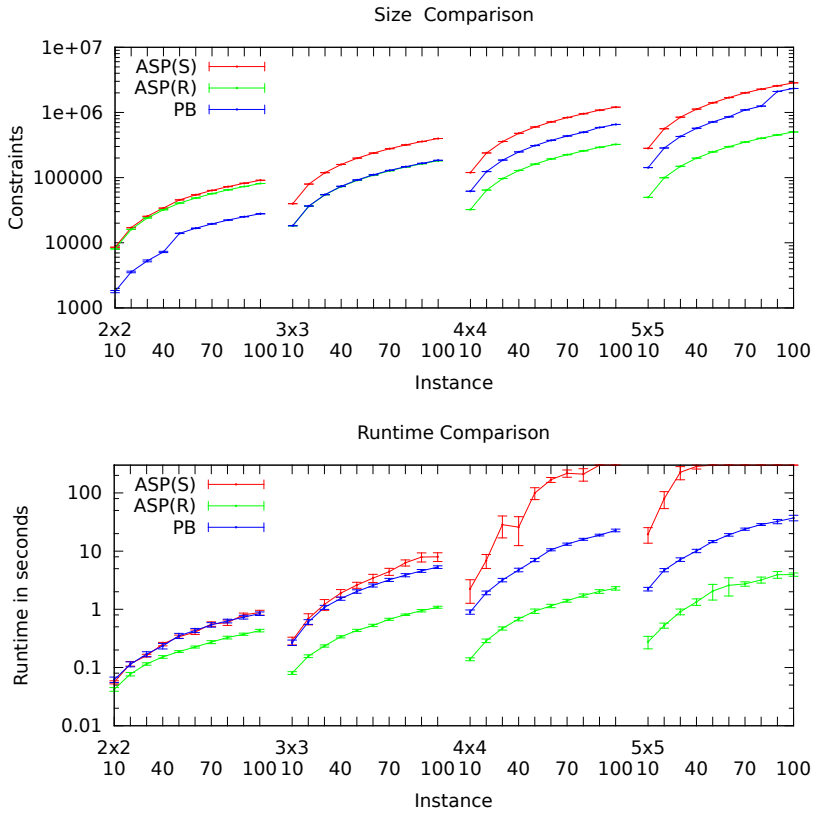

Fig. 6: Average numbers of constraints and runtimes in seconds for mesh networks of varying sizes and task numbers.

than ASP(S) and PB, inducing almost one order of magnitude fewer constraints than the latter. Compared to this, the observation that the PB-based approach requires fewer constraints for the smallest instances (with average runtimes in split seconds) is negligible. The corresponding average runtimes in the right chart tightly correlate to representation sizes. While ASP(S) can still cope with small instances, it is drastically worse than PB and ASP(R) from mesh size 4x4 on, and it times out on all instances of size 5x5 with 50 or more tasks. However, as the average runtimes of PB and ASP(R) (the latter again by about one order of magnitude smaller than the former) show, even the larger instances are manageable by means of preprocessing (PB) or avoiding step counting (ASP(R)).

For investigating the further scaling behavior, we applied ASP(R) to larger meshes, using fixed ratios between the number of tasks and available CPUs as shown in Figure 7. (We here omit ASP(S) and PB in view of poor solving performance or long instance generation time, respectively.) While 6x6 mesh networks could easily be solved within seconds, we encountered first timeouts (6 out of 16) on instances of size 7x7 along with 245 process tasks (five per CPU). However, some instances (14 or 3, respectively, out of

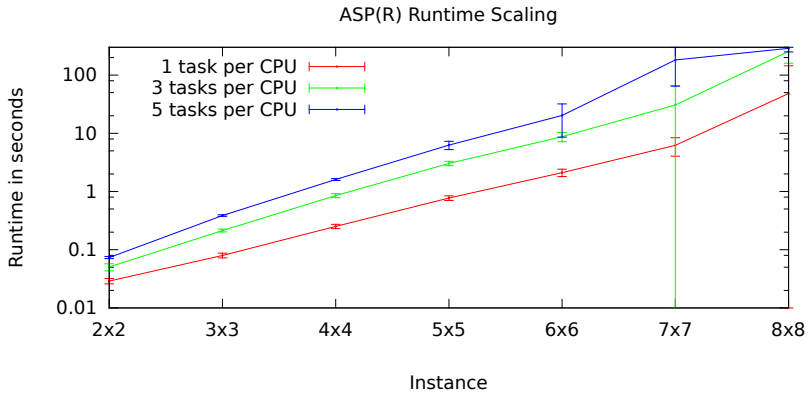

Fig. 7: Average runtimes in seconds for mesh networks of scaled up sizes.

16) of size 8x8 could still be solved within the time limit of 300 seconds when given one or three tasks per CPU, i.e., 64 or 192 tasks in total. Since the problem representation size (cf. numbers of constraints) is linear in the input for ASP(R), the timeouts on large instances are explained by increasing variance of solving performance in view of randomness in problem generation. Regarding the robustness of solving, we conjecture that it can be improved by including domain knowledge in ASP encodings, somewhat similar to simplifications performed by the PB generator, yet specified declaratively by rules rather than implemented by special-purpose procedural components.

6 Conclusion

We proposed a novel approach to system synthesis using ASP. While our naive step-oriented ASP encoding cannot compete with the sophisticated PB/SAT-based approach in [10], the succinct ASP formulation of reachability, as required in multi-hop routing, outperforms previous approaches when applied to densely connected (mesh) networks, providing vast routing options. Such performance gains are made possible by considerably smaller design space representations and accordingly reduced search efforts. Given that ASP solvers like *clasp* also support optimization, the presented ASP approach could be extended to linear and, with some adaptions, even be utilized for non-linear optimization, as previously performed in design space exploration via evolutionary algorithms [11]. At user level, the declarative first-order modeling language of ASP facilitates prototyping as well as adjustment of ASP solutions for new or varied application scenarios, making it a worthwhile alternative to purely propositional formalisms.

Acknowledgments. This work was partially funded by DFG grant SCHA 550/XY-Z and SCHA 550/9-1.

References

1. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
2. A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
3. T. Blickle, J. Teich, and L. Thiele. System-level synthesis using Evolutionary Algorithms. *J. Design Automation for Embedded Systems*, 3(1):23–58, 1998.
4. S. Borkar. Thousand core chips: a technology perspective. In *Proc. of DAC '07*, pages 746–749, 2007.
5. M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):105–124, 2011.
6. M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`.
7. M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.
8. Christian Haubelt, Jürgen Teich, Rainer Feldmann, and Burkard Monien. SAT-Based Techniques in System Design. In *Proc. of DATE '03*, pages 1168–1169, 2003.
9. H. Ishebabi, P. Mahr, C. Bobda, M. Gebser, and T. Schaub. Answer set vs integer linear programming for automatic synthesis of multiprocessor systems from real-time parallel programs. *Journal of Reconfigurable Computing*, 2009. Article ID 863630.
10. Martin Lukasiewycz at al. Combined System Synthesis and Communication Architecture Exploration for MPSoCs. In *Proc. of DATE '09*, pages 472–477. IEEE Computer Society, 2009.
11. Martin Lukasiewycz et al. Efficient symbolic multi-objective design space exploration. In *Proc. of ASP-DAC '08*, pages 691–696, 2008.
12. Sandeep Neema. *System Level Synthesis of Adaptive Computing Systems*. PhD thesis, Vanderbilt University, Nashville, Tennessee, May 2001.
13. Ralf Niemann and Peter Marwedel. An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming. *Design Automation for Embedded Systems*, 2(2):165–193, 1997.