ASP-based Knowledge Representations for IoT Configuration Scenarios

Alexander Felfernig[1] and **Andreas Falkner**[2] and
Müslüm Atas[1] and **Seda Polat Erdeniz**[1] and **Christoph Uran**[1] and **Paolo Azzoni**[3]

Abstract. The purpose of this paper is to introduce basic application scenarios for configuration technologies in Internet of Things (IoT) product domains. We show how to represent configuration knowledge in the domain of *smart homes* on the basis of Answer Set Programming (ASP). In this context, we introduce different configuration model elements and constraint types and show their corresponding ASP representation in a way that is also useful for ASP beginners. We conclude the paper with a discussion of open issues for future work.

1 Introduction

Configuration is one of the most successfully applied Artificial Intelligence technologies [7, 18]. It is a specific type of design activity where a product is configured on the basis of a set of already defined component types and corresponding constraints that restrict the way in which component instances can be combined. A *configuration task* is defined in terms of a generic product structure, a corresponding set of constraints, and a set of requirements (often also denoted as customer requirements) that additionally restrict the set of possible solutions. A solution (configuration) for a configuration task is represented by a set of component instances, their connections and attribute settings which altogether are consistent with the constraints and requirements included in the configuration task definition.

There is a multitude of application domains for knowledge-based configuration – example domains are the automotive sector, financial services, operating systems, software product lines, and railway interlocking systems [7]. Configuration technologies nowadays become increasingly popular in different kinds of Internet of Things (IoT) [1] scenarios. The Internet of Things is an emerging paradigm that envisions a networked infrastructure which enables the interconnection of devices (things) anyplace and anytime. In the IoT context, configurators can be applied, for example, to the identification of ramp-up configurations (self-configuration), i.e., to figure out which components (potential software and hardware) are needed in a certain IoT setting. Configuration technologies can also be applied during runtime where, for example, a configurator helps to identify pareto-optimal configurations of communication protocols with regard to criteria such as *performance* and *cost of data transfer*.

The size and complexity of configuration problems in the IoT domain often does not allow the application of basic configuration knowledge representation and reasoning such as constraint satisfaction [22]. Smart homes often consist of hundreds or even thousands of different components and constraints – such scenarios are in the need of a component-oriented knowledge representation that is easy to use and maintain [7, 13]. Open source constraint-based approaches do not support such a representation and existing component-oriented commercial environments are based on proprietary knowledge representations with limitations also in terms of standardization. An alternative to constraint-based knowledge representations especially useful for large and complex configuration domains is Answer Set Programming (ASP) [12, 17]. ASP supports the definition of component hierarchies and related constraints in a declarative way. Potential component instances have to be predefined, i.e., in its basic form ASP does not support pure component generation during runtime.

There exist a couple of research contributions related to the application of answer set programming in the configuration context. Soininen and Niemelä [17] can be considered as pioneers who first showed the application of ASP to represent and solve configuration tasks. A resulting configuration environment is presented, for example, in [7, 21]. An object-oriented layer to answer set programs has been introduced by [4]. In this work, configuration tasks can be represented on an object-oriented level without being forced to take into account specific details of ASP-based configuration knowledge representations. Thus, this work can be seen as a contribution to improve the applicability of ASP technologies especially in terms of reducing efforts related to knowledge base development and maintenance. Feature model related ASP representations are introduced in [15]. An approach to the testing of object-oriented models on the basis of ASP is introduced in [6]; in this context it is shown how UML-based configuration knowledge representations can be represented in ASP and how positive and negative test cases can be represented and included for the purpose of supporting unit tests on knowledge bases. Friedrich et al. [8] introduce an approach to re-configuration in ASP – in this context, a reconfiguration can be considered as a set of changes to an already existing configuration such that new requirements are taken into account. Finally, Teppan et al. [20] introduce a hybrid approach that integrates constraint solving with ASP. A major advantage of this integration is that the grounding bottleneck in answer set programming can be transformed into a more efficiently solvable search problem in constraint programming. The major focus of this paper are ASP-based knowledge representations. For an overview of different further approaches to configuration knowledge representation we refer to [13].

The contributions of this paper are the following. First, we introduce ASP-based configuration knowledge representations in the

[1] Institute for Software Technology, Graz University of Technology, Inffeldgasse 16b/2, A-8010 Graz, email:{a.felfernig, muesluem.atas, spolater, christoph.uran}@ist.tugraz.at
[2] Siemens AG Österreich, email:andreas.a.falkner@siemens.com
[3] Eurotech Group, Italy, email: paolo.azzoni@eurotech.com

context of Internet Of Things (IoT) scenarios. Second, our aim is to provide easy to understand examples (for ASP newbies) of how to represent configuration knowledge in ASP and also to show limitations of ASP knowledge representations. Third, we discuss different issues for future research that will help to accelerate a broad application of ASP technologies in knowledge-based configuration.

The remainder of this paper is organized as follows. In Section 2 we discuss IoT-related configuration domains and introduce a smart home configuration model which is used as working example throughout the paper. In Section 3 we show how to translate individual model elements into a corresponding ASP-based representation. In this context we also sketch how ASP solvers operate to determine a solution for a configuration task (Section 4). In Section 5 we sketch the role of ASP solving in our IoT-related research project. The paper is concluded with a discussion of open research issues (Section 6).

2 IoT Domains and Configuration Models

In the following we provide an overview of IoT domains where the application of ASP-based configuration technologies is reasonable. In this context, we introduce a simplified configuration model from the domain of smart homes in order to show different facets of ASP-based configuration knowledge representations.

Air Pollution Monitoring. Air pollution monitoring systems help to ensure healthy living conditions, for example, in cities. An issue in this context is the distribution of sensors in a city topology that assures a representative collection of measurement data. This data is analyzed on the basis of different types of learning algorithms that help to figure out in which contexts which actions have to be triggered. Examples of related actions are a general warning to leave the house, reduced speed limits on highways, recommendations to groups (e.g., schools) in terms of the maximum time that should be spent outdoors, and warnings regarding the malfunctioning of filter equipments in industrial production. In air pollution monitoring, configuration technologies can be used to select the type and placement of sensors given a specific topology (e.g., a topology of a city) and also to select the types of algorithms that should be used for data analysis in certain contexts.

Health Monitoring. Health monitoring solutions can be based on different types of data that can be used to determine recommendations related to factors such as eating behavior, sports activities, sleeping times, and also data about the body condition. Many tools already allow the manual entering of consumed food, however, in future scenarios such information will be available on the basis of standardized data exchange protocols. Measurement of sports activities and sleeping time is already included in many commercial solutions. Finally, detailed information about the physical condition of a person is not taken into account in many of the existing tools. In such scenarios, configuration technologies can be used to parametrize the underlying algorithms, for example, recommended heart rates when doing physical practices depend on the age, gender, and weight of a person (and further physical parameters). Whether specific food items can be recommended or not depends, for example, on potential allergies of a person. Finally, especially in group sports (e.g. football or tennis), the type of training also depends on the participating persons. For example, if three persons are participating in a tennis training session and one person has a bad physical condition, this has an impact on the selection of exercise units for this group.

Energy Production and Management. Energy production is in the need of configuration technologies in various scenarios, for example, wind turbines must be configured and parametrized in order to be able to maximize energy production in a certain environment. Knowledge about where and when a higher amount of energy will be needed can trigger a corresponding reconfiguration of the load factor of water reservoirs. In the context of private energy production, configuration technologies can help to rearrange energy consumption times, for example, when to recharge the electric car or when to activate the washing machine. Especially in the context of energy management in buildings, reconfiguration technologies can play a role by supporting the change of building parameters depending on given environmental data such as temperature, weather conditions, and forecasts.

Enhanced Retail Services. In-store shopping is based on specific distributions of sensors and other devices such as information displays. During the ramp-up phase of such an application it has to be assured that customer location sensors are distributed in a reasonable fashion and information displays are positioned in such a way that the information can be easily seen by customers. In such scenarios, configuration technologies can be applied in order to determine the amount of sensors needed, the positioning of information displays, and also to determine the layout of the whole shop depending on the product assortment that should be offered to a customer.

Animal Monitoring. There exist a couple of scenarios where information about animal locations and information about the physical condition of animals is important. For example, in wildlife scenarios where animals are spread over huge and not accessible areas, it is important to provide an infrastructure for animal monitoring that is not based on physical presence of human administrators. In such contexts, for example, different types of drones can be used to support data collection. Depending on the region size and topography and requirements regarding the amount of data to be collected, drones have to be configured in a way that optimizes the trade-off between energy consumption, range, and support of the defined data collection requirements. In such scenarios, configuration technologies can be useful to support the complete configuration of the needed data collection equipment.

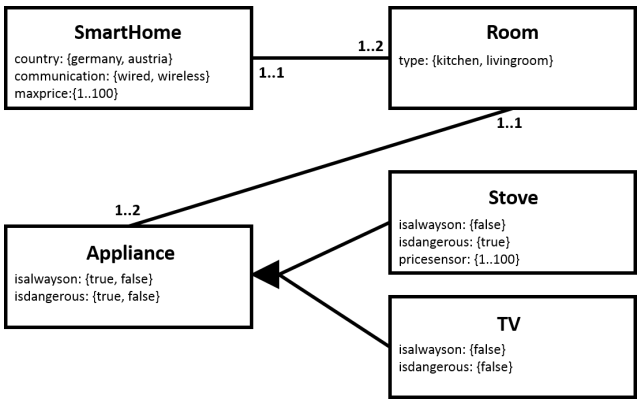

Figure 1. Simplified configuration model (reduced #component types, #attributes, domains, and multiplicities) of a *smart home* used for demonstration purposes. Additional constraints are presented in our discussion of ASP-based configuration knowledge representations.

Smart Homes. A simplified smart home configuration model is depicted in Figure 1. Smart homes [14] include functionalities for actively supporting persons in their daily life. Examples thereof are *intelligent light management* that allows to (semi-automatically) adapt the illumination of rooms depending on the time of the day and sea-

son, *energy management* that supports intelligent air conditioning for whole buildings, *security management* (e.g., when nobody is at home), and functionalities related to the support of *ambient assisted living* scenarios with related functionalities such as automated fall detection. In such scenarios, configuration technologies can be used to design in detail which smart home hardware and software components (e.g., sensors, actuators, and apps) have to be installed in which part of the building. The smart home model depicted in Figure 1 will serve as working example in this paper to demonstrate ASP-based configuration knowledge representations.

3 Configuration Knowledge Representation in ASP

In the following we will show how to represent configuration knowledge for a simplified configuration model from the domain of smarthomes (see Figure 1).[4]

(a) *Potential Component Instances*. In the ASP context, all decision variables have to be pre-specified. In our example model shown in Figure 1, three different component types are included which are represented by the predicate names *smarthome*, *room*, and *appliance* (with the subtypes *stove* and *room*). For these component types we have to specify the maximum amount of instances that can be part of a related smarthome configuration (see Figure 2), i.e., one *smarthome* (e.g., *psmarthome*(1) denotes a potential instance of *smarthome* with id 1), two instances of *room*, and 4 instances of each subtype of *appliance*. The integer number arguments (e.g., *proom(2;3)*) in the ASP facts serves as a unique key to distinguish the (potential) instances.

```
psmarthome(1).
proom(2;3).
pstove(4;5;6;7).
ptv(8;9;10;11).
```

Figure 2. Definition of potential smart home component instances using Answer Set Programming (ASP) notation. For example, *proom(2;3)* denotes two potential instances of type *room*.

(b) *Component Types and Instances*. For a configuration, it has to be decided which of the potential instances shall be included. Therefore we establish an association of the aforementioned definitions of potential instances with the "real" component instances that can be included in a configuration (see Figure 3). For each potential instance it has to be decided whether to include this as an instance in a configuration or not, for example in our configuration knowledge base, each potential room instance can be part of a configuration or not (this is specified by the corresponding lower and upper bounds of the generation rule).

```
0{ smarthome(X) }1 :- psmarthome(X).
0{ room(X) }1 :- proom(X).
0{ stove(X) }1 :- pstove(X).
0{ tv(X) }1 :- ptv(X).
```

Figure 3. Definition of component types (in ASP). For example, each potential instance of type room (i.e., *proom(X)*) can be part of a configuration as *room(X)*.

(c) *Generalization Hierarchies*. Generalization hierarchies allow to further categorize different component types (see Figure 4). For demonstration purposes, we represent *stove* and *tv* as (disjunctive) subtypes of the component type *appliance* – an alternative to introducing a *type* attribute similar as for *room*. We also include a rule that assures that instances of appliances are instantiated to *stove* or *tv*. Note that disjunctiveness between subtypes is assured by definition (see Figures 2 and 3).

As we are not interested in incomplete configurations (e.g., instances of *appliance* that are not refined to *stove* or *tv*), we add a constraint that ensures that each *appliance* instance is either refined to a *stove* instance or a *tv* instance. This is only necessary when users are allowed, for example, to include component instances represented by corresponding facts (e.g., *appliance(4)*).

```
appliance(X) :- stove(X).
appliance(X) :- tv(X).
:- appliance(X), not stove(X), not tv(X).
```

Figure 4. Defining generalization hierarchies (in ASP). For example, each *stove* is an *appliance* and each *tv* is an *appliance*, and vice-versa, each *appliance* is either a *stove* or a *tv*.

(d) *Attributes*. For the defined component type attributes, we have to introduce attribute domain definitions (see Figure 5).

```
dommaxprice(1..100).
domcountry(germany;austria).
domcommunication(wired;wireless).
domtype(kitchen;livingroom).
domisalwayson(true;false).
domisdangerous(true;false).
dompriceofsensor(60).
```

Figure 5. Attribute domain definitions (in ASP). For example, *dommaxprice* represents an attribute that will be used to specify the maximum price of a *smarthome* solution. For simplicty, we only include price information related to *stoves* – see also Figure 6.

Attribute domain definitions have to be associated with the corresponding component types, for example, the domain definition *domcountry* of the attribute *country* has to be associated with the component type *smarthome* (see Figure 6). With ASP choice rules we enforce that each instance has exactly one domain value for each of its attributes. Generalization is covered in a natural way: see the right-hand-side of the last two rules in the figure. As attributes are created only for existing instances (but not for potential instances), spurious solutions (such as arbitrary attribute settings for unused potential instances and their combination) are avoided.

```
1{country(X,Y):domcountry(Y)}1:- smarthome(X).
1{communication(X,Y):domcommunication(Y)}1:- smarthome(X).
1{maxprice(X,Y):dommaxprice(Y)}1:-smarthome(X).
1{type(X,Y):domtype(Y) }1:- room(X).
1{isalwayson(X,Y):domisalwayson(Y)}1:- appliance(X).
1{isdangerous(X,Y):domisdangerous(Y)}1:- appliance(X).
1{priceofsensor(X,Y):dompriceofsensor(Y)}1:- stove(X).
```

Figure 6. Associating attributes with component types (in ASP). For example, *country* is an attribute associated with *smarthomes*. On an instance level, attribute instances are only generated if corresponding component instances exist, i.e., attribute instances are only created when necessary.

If domain definitions are reduced in subcomponent types, this can be expressed in a corresponding ASP rule, for example, *isdangerous(X,false):- tv(X).* expresses the fact that a *tv* set is not considered as a dangerous appliance (see Figure 7).

(e) *Associations and Multiplicities*. Associations between component types on the model level are represented in terms of binary pred-

[4] For demonstration purposes we use the syntax of the *clingo* environment (see potassco.sourceforge.net).

```
isalwayson(X,false) :- stove(X).
isdangerous(X,true) :- stove(X).
isalwayson(X,false) :- tv(X).
isdangerous(X,false) :- tv(X).
```

Figure 7. Reducing ASP attribute domain definitions, for example, in generalization hierarchies. For example, the *isdangerous* attribute of type *appliance* is reduced to *true* if the corresponding component is a *stove*.

icates on the ASP level (see Figure 8). We use ASP rules to define potential links, e.g. *smarthomeroom*, with the allowed minimum and maximum multiplicities of the association.

```
1{ smarthomeroom(X,Y): room(Y) }2 :- smarthome(X).
1{ smarthomeroom(Y,X): smarthome(Y) }1 :- room(X).
1{ roomappliance(X,Y): appliance(Y) }2 :- room(X).
1{ roomappliance(Y,X): room(Y) }1 :- appliance(X).
```

Figure 8. Defining associations and corresponding multiplicities (in ASP). For example, each *smarthome* has 1-2 associated components of type *room*.

(f) *Incompatibility Constraints*. Such constraints typically specify incompatibilities regarding the combination of specific component types or simply combinations of incompatible attribute values. An example of an incompatibility is represented by the following constraint that expresses the fact that a *tv* should not be situated in a room of type *kitchen* (see Figure 9).

```
:- roomappliance(X,Y), type(X,kitchen), tv(Y).
```

Figure 9. Defining incompatibility constraints (in ASP). For example, no *tv* should exist in a *kitchen*.

(g) *Requires Constraints*. Requirements relationships describe situations where the integration of a certain component or the selection of a certain attribute value also requires the integration/selection of further component types/attribute values. An example constraint is the following: *smarthomes* in *Austria* must have at least two rooms (the corresponding ASP representation is shown in Figure 10).

```
2{ smarthomeroom(X,Y): room(Y) }2 :-
    smarthome(X), country(X,austria).
```

Figure 10. Defining requires constraints (in ASP). For example, *smarthomes* in *Austria* include at least two *rooms*.

(h) *Resource Constraints*. Resource constraints specify producer and consumer relationships, for example, a resource (*producer*) could be the money available for the smarthome (*maxprice* specified by the customer) and the consumers could be the installed sensors (represented by the attribute *priceofsensor*). A corresponding resource constraint could indicate that the sum of the prices of all sensors must not exceed the upper price limit specified by the customer (attribute *maxprice*). An implementation of a resource constraint in ASP is shown in Figure 11.

(i) *Navigation Constraints*. ASP allows to represent complex constraints which require navigation between instances. For example, a stove in one room excludes further stoves in other rooms of the same smarthome (see Figure 12). This can also be interpreted as a further example of an incompatibility constraint (see Figure 9). Even recursively defined predicates can be used in such constraints such that transitive closures (e.g. reachability in graphs) and constraints

```
sensorprice(T) :- T = #sum{ PR, A : priceofsensor(A, PR)}.
:- smarthome(X), maxprice(X,Y), sensorprice(P), P > Y.
```

Figure 11. Defining resource constraints (in ASP). For example, the price of a *smarthome* (represented by *sensorprice* only) must not exceed the *maxprice* defined by the customer.

on them can be expressed. This is a modeling advantage compared to standard constraint solvers.

```
:- stove(A1), roomappliance(R1,A1), smarthomeroom(H,R1),
   R1 != R2,
   smarthomeroom(H,R2), roomappliance(R2,A2), stove(A2).
```

Figure 12. Defining navigation constraints (in ASP). For example, two different *rooms* with a *stove* must not be part of the same *smarthome* configuration. In this context, R1 != R2 assures that two different *rooms* are analyzed with regard to the inclusion of a *stove*.

(j) *Example Customer Requirements*. Having defined the whole configuration knowledge base, customers can specify their requirements with regard to a corresponding smarthome configuration as facts and even more generally as constraints (see Figure 13). Examples of such customer requirements are: *the smarthome installation will be located in Austria* and *no dangerous appliances should be installed* (see the following constraints).

```
smarthome(1).
country(1,austria).
:- appliance(X), isdangerous(X,true).
```

Figure 13. Defining requirements (in ASP). For example, the *smarthome* should be in *austria* and no dangerous *appliances* should be included.

4 ASP Solving and Limitations

Classical ASP solvers [9] work in two steps: (1) *grounding* which translates the ASP program to a variable-free format and (2) propositional (SAT) *solving* of the grounded program. Figure 14 shows a reduced version of our *smarthome* configuration knowledge base.

```
% potential instances
psmarthome(1). proom(2;3;4).

% attribute domain definitions
domcountry(germany;austria). domtype(kitchen;livingroom).

% definition / generation of instances
0{ smarthome(X) }1 :- psmarthome(X).
0{ room(X) }1 :- proom(X).

% associating attributes with component types
1{ country(X,Y): domcountry(Y) }1 :- smarthome(X).
1{ type(X,Y): domtype(Y) }1 :- room(X).

% definition / generation of associations
1{ smarthomeroom(X,Y): room(Y) }2 :- smarthome(X).
1{ smarthomeroom(Y,X): smarthome(Y) }1 :- room(X).

% further constraints
:- smarthome(X), country(X,austria),
   not 2{ smarthomeroom(X,Y): room(Y) }2.

% customer requirements
room(2). type(2,kitchen).
```

Figure 14. Restricted version of example *smarthome* configuration model.

The knowledge base in Figure 14 is a subset of the class diagram in Figure 1. It includes a component type *smarthome* with the at-

tribute *country* and a component type *room* with the attribute *type*. A *smarthome* can have 1 or 2 *rooms* but Austrian *smarthomes* must have two rooms (this is defined in terms of an additional constraint). One potential instance is defined for *smarthome* and three potential instances are defined for *room*. The requirements specify the inclusion of a *room of type kitchen*.

The *grounding* results are shown in Figure 15 – the relationship to the knowledge base of Figure 14 is explained in terms of comments. *ASP facts* of the original knowledge base are also represented as facts in the grounded knowledge base. Variables in rules are removed and the rules are duplicated accordingly, for example, three rules are generated for the three possible room instances: i.e., *proom(X)* in the second rule for generation of instances is replaced with each of the three facts for potential instances and the head of the rule is replaced accordingly. Furthermore, the *count* aggregate is represented in its standard form instead of just curly brackets.

```
% potential instances
psmarthome(1). proom(2). proom(3). proom(4).

% attribute domain definitions
domcountry(germany). domcountry(austria).
domtype(kitchen). domtype(livingroom).

% definition / generation of instances
0<=#count{1,0,smarthome(1):smarthome(1)}<=1.
0<=#count{1,0,room(2):room(2)}<=1.
0<=#count{1,0,room(3):room(3)}<=1.
0<=#count{1,0,room(4):room(4)}<=1.

% associating attributes with component types
1<=#count{1,0,country(1,germany):country(1,germany);
          1,0,country(1,austria):country(1,austria)
         }<=1:- smarthome(1).
1<=#count{1,0,type(2,kitchen):type(2,kitchen);
          1,0,type(2,livingroom):type(2,livingroom)
         }<=1:- room(2).
1<=#count{1,0,type(3,kitchen):type(3,kitchen);
          1,0,type(3,livingroom):type(3,livingroom)
         }<=1:-room(3).
1<=#count{1,0,type(4,kitchen):type(4,kitchen);
          1,0,type(4,livingroom):type(4,livingroom)
         }<=1:-room(4).

% definition / generation of associations
1<=#count{
    1,0,smarthomeroom(1,2):smarthomeroom(1,2):room(2);
    1,0,smarthomeroom(1,3):smarthomeroom(1,3):room(3);
    1,0,smarthomeroom(1,4):smarthomeroom(1,4):room(4)
         }<=2:-smarthome(1).

1<=#count{1,0,smarthomeroom(1,2):
            smarthomeroom(1,2):smarthome(1)}<=1:-room(2).
1<=#count{1,0,smarthomeroom(1,3):
            smarthomeroom(1,3):smarthome(1)}<=1:-room(3).
1<=#count{1,0,smarthomeroom(1,4):
            smarthomeroom(1,4):smarthome(1)}<=1:-room(4).

% further constraints
:- smarthome(1); country(1,austria); not 2<=#count
  {1,0,smarthomeroom(1,2):smarthomeroom(1,2):room(2);
   1,0,smarthomeroom(1,3):smarthomeroom(1,3):room(3);
   1,0,smarthomeroom(1,4):smarthomeroom(1,4):room(4)}<=2.

% customer requirements
room(2). type(2,kitchen).
```

Figure 15. Grounded version of restricted *smarthome* configuration model.

In general, grounding can lead to extremely large knowledge bases, especially if rules or constraints entail many variables with a large domain. The domain sizes are multiplied which leads to exponential growth in the number of variables in the worst case. This shows one of the weaknesses of answer set programs - they are not well suited for problems with *large integer domains or floating point numbers*. Ways to deal with this issue is to combine answer set programming with constraint solving techniques (see, e.g., [11, 20]) and lazy grounding (see, e.g., [3]).

Solving an ASP results in answer sets (solutions). Each solution must be well-founded (i.e., derived from the given facts) and consistent (i.e. not violating any constraint). Figure 16 shows all solutions (answer sets) derived from the example in Figure 14.

As the customer prefers *room 2* and *country austria* requires 2 rooms, there is only one answer set (*Answer: 1*) with one room - its type is *kitchen* (as specified by the customer requirements) and the *country* is *germany*. Sketch of the reasoning steps: *room(2)* can be seen as a propositional variable with truth value *TRUE*. It appears in the body of the second rule for generation of associations and that body contains no other variables. Therefore the head is evaluated and requires exactly one variable in the count aggregate to be set to *TRUE*. The only one is *smarthomeroom(1,2)* and it is founded if *smarthome(1)* is generated by the first count aggregate for generation of instances. Therefore, it derives the facts *smarthomeroom(1,2)* and *smarthome(1)*. For *country*, there are exactly two alternatives, but only *germany* does not lead to a constraint violation. Therefore *country(1,germany)* is added as a fact (i.e. assigned truth value *TRUE* in the SAT view of the reasoning process). As no other variables need to be set (all count aggregates are fulfilled), we have the first solution.

All other solutions derive a second room. This allows all combinations of the alternatives for *type (kitchen, livingroom)*, *country (germany, austria)* and *identifier (3, 4)*. One can imagine that the corresponding multiplication of alternatives can lead to tremendously many solutions. At least concerning the identifiers, the solutions are equivalent (i.e. there is no relevant difference between answer sets 2 to 5 and answer sets 6 to 9). In order to reduce the search space for such unneeded solutions, symmetry breaking techniques [2] and search heuristics [10] can be used.

```
- Answer: 1 -
smarthome(1) country(1,germany)
room(2) type(2,kitchen)
smarthomeroom(1,2)
- Answer: 2 -
smarthome(1) country(1,germany)
room(2) type(2,kitchen) room(4) type(4,livingroom)
smarthomeroom(1,2) smarthomeroom(1,4)
- Answer: 3 -
smarthome(1) country(1,austria)
room(2) type(2,kitchen) room(4) type(4,livingroom)
smarthomeroom(1,2) smarthomeroom(1,4)
- Answer: 4 -
smarthome(1) country(1,germany)
room(2) type(2,kitchen) room(4) type(4,kitchen)
smarthomeroom(1,2) smarthomeroom(1,4)
- Answer: 5 -
smarthome(1) country(1,austria)
room(2) type(2,kitchen) room(4) type(4,kitchen)
smarthomeroom(1,2) smarthomeroom(1,4)
- Answer: 6 -
smarthome(1) country(1,germany)
room(2) type(2,kitchen) room(3) type(3,kitchen)
smarthomeroom(1,2) smarthomeroom(1,3)
- Answer: 7 -
smarthome(1) country(1,austria)
room(2) type(2,kitchen) room(3) type(3,kitchen)
smarthomeroom(1,2) smarthomeroom(1,3)
- Answer: 8 -
smarthome(1) country(1,germany)
room(2) type(2,kitchen) room(3) type(3,livingroom)
smarthomeroom(1,2) smarthomeroom(1,3)
- Answer: 9 -
smarthome(1) country(1,austria)
room(2) type(2,kitchen) room(3) type(3,livingroom)
smarthomeroom(1,2) smarthomeroom(1,3)
```

Figure 16. Solutions for *smarthome* knowledge base of Figure 15.

Figure 17 shows symmetry breaking constraints which force the solver to use identifiers from lowest to highest. In the knowledge base defined by the ASP entries of Figure 2–13, the number of answer sets would be reduced from 8.400 to 2.800 if the symmetry breaking constraints are taken into account. Adding constraint *:- room(X),*

proom(Y), X>Y, not room(Y). to Figure 14 would reduce the number of answer sets from 9 to 5 in Figure 16.

```
:- room(X), proom(Y), X>Y, not room(Y).
:- stove(X), pstove(Y), X>Y, not stove(Y).
:- tv(X), ptv(Y), X>Y, not tv(Y).
```

Figure 17. Symmetry breaking constraints for the entries of Figures 2–13.

5 AGILE Configuration Technologies

The configuration knowledge representations discussed in this paper are applied within the scope of the European Union project AGILE[5] that focuses on the development of recommendation and configuration technologies for IoT gateways. Within AGILE, configuration technologies are applied to support different kinds of ramp-up scenarios, i.e., initial setups of IoT gateways infrastructures entailing the needed hardware and software components. Furthermore, AGILE supports runtime configuration and reconfiguration, for example, in terms of optimizing the usage of data exchange protocols with regard to optimality criteria such as economy and efficiency. The basis for AGILE configuration solutions in ramp-up domains is the *clingo* environment. In AGILE, we are especially focusing on improving the performance of constraint-based reasoning and model-based diagnosis that are both supporting technologies also in the context of answer set programs [16, 20].

6 Research Issues

There are still a couple of research challenges to be tackled to make ASP-based configuration more applicable and performant. Graphical configuration knowledge representations and a corresponding automated translation into ASP-based representations will help to improve knowledge engineering processes. An automated generation of ASP knowledge bases from object-oriented product topologies has already been proposed in [5]; translation routines for standard constraints such as *requires*, *excludes*, and *resources* would help to further increase knowledge engineering efficiency. Improving constraint answer set programming or finding new ways of integrating ASP and CSP could lead to a full exploitation of the different advantages of both paradigms. In order to solve large-sized problems, lazy grounding and heuristics must be combined and improved (related work is reported, e.g., in [23]). All means to reduce problem sizes of ASPs should be exploited, for example, precise estimation of the number of needed instances (see [19]).

7 Conclusions

In this paper we give an overview of basic applications of configuration technologies in Internet of Things (ioT) scenarios. In this context we show how to apply answer set programming (ASP) techniques to represent and solve configuration problems. ASP is a logic-based approach and well-suited for a component-oriented representation of configuration tasks. This capability is extremely useful especially in large and complex product domains. In order to provide a basic reference for ASP beginners, we show how to represent most representative constraints in ASP.

[5] agile-iot.eu.

ACKNOWLEDGEMENTS

The work presented in this paper has partially been conducted within the scope of the Horizon 2020 Project AGILE (Adoptive Gateways for dIverse MuLtiple Environments, 2016–2018).

REFERENCES

[1] L. Atzori, A. Iera, and G. Morabito, 'The Internet of Things: A Survey', *Computer Networks*, **54**(15), 2787–2805, (2010).

[2] C. Drescher, O. Tifrea, and T. Walsh, 'Symmetry-breaking answer set solving', in *ICLP10 Workshop on Answer Set Programming and Other Computing Paradigm*, (2010).

[3] T. Eiter, T. Kaminski, C. Redl, and A. Weinzierl, 'Exploiting Partial Assignments for Efficient Evaluation of Answer Set Programs with External Source Access', in *25th International Joint Conference on Artificial Intelligence (IJCAI-16)*, pp. 1058–1065, New york, NY, USA.

[4] A. Falkner, A. Ryabokon, G. Schenner, and K. Shchekotykhin, 'OOASP: Connecting Object-Oriented and Logic Programming', in *13th International Conference on Logic Programming and Nonmonotonic Reasoning*, pp. 332–345, Lexington, KY, USA, (2015).

[5] A. Falkner, A. Ryabokon, G. Schenner, and K. Shchekotykhin, 'OOASP: connecting object-oriented and logic programming', *CoRR*, (2015).

[6] A. Falkner, G. Schenner, G. Friedrich, and A. Ryabokon, 'Testing Object-Oriented Configurators With ASP', in *ECAI 2012 Workshop on Configuration*, pp. 21–26, Montpellier, France, (2012).

[7] A. Felfernig, L. Hotz, C. Bagley, and J. Tiihonen, *Knowledge-based Configuration: From Research to Business Cases*, Elsevier/Morgan Kaufmann Publishers, 1st edn., 2014.

[8] G. Friedrich, A. Ryabokon, A. Falkner, A. Haselböck, G. Schenner, and H. Schreiner, '(Re)configuration using Answer Set Programming', in *Workshop on Configuration*, pp. 17–25, Barcelona, Spain, (2011).

[9] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, *Answer Set Solving in Practice*, Morgan & Claypool Publishers, 1st edn., 2012.

[10] M. Gebser, B. Kaufmann, J. Romero, R. Otero, T. Schaub, and P. Wanko, 'Domain-Specific Heuristics in Answer Set Programming', in *27th AAAI Conf. on AI*, pp. 350–356, Bellevue, WA, USA.

[11] M. Gebser, A. Ryabokon, and G. Schenner, 'Combining heuristics for configuration problems using answer set programming', in *13th International Conference on Logic Programming and Nonmonotonic Reasoning*, pp. 384–397, Lexington, KY, USA, (2015).

[12] M. Gelfond and V. Lifschitz, 'The stable model semantics for logic programming', in *5th International Conference of Logic Programming (ICLP'88)*, pp. 1070–1080, (1988).

[13] L. Hotz, A. Felfernig, M. Stumptner, A. Ryabokon, C. Bagley, and K. Wolter, 'Configuration Knowledge Representation and Reasoning', in *Knowledge-Based Configuration: From Research to Business Cases*, 41–72, (2014).

[14] G. Leitner, A. Fercher, A. Felfernig, K. Isak, S. Polat Erdeniz, A. Akcay, and M. Jeran, 'Recommending and configuring smart home installations', in *Workshop on Configuration*, pp. 17–22, (2016).

[15] V. Myllärniemi, J. Tiihonen, M. Raatikainen, and A. Felfernig, 'Using Answer Set Programming for Feature Model Representation and Configuration', in *Workshop on Configuration*, pp. 1–8, (2014).

[16] K. Shchekotykhin, 'Interactive Query-Based Debugging of ASP Programs', in *29th AAAI Conf. on AI*, pp. 1597–1603, Austin, Texas, USA.

[17] T. Soininen and I. Niemelä, 'Developing a declarative rule language for applications in product configuration', in *PADL*, pp. 305–319, (1998).

[18] M. Stumptner, 'An Overview of Knowledge-based Configuration', *AI Communications*, 111–125, (1997).

[19] R. Taupe, A. Falkner, and G. Schenner, 'Deriving tighter component cardinality bounds for product configuration', in *18 th International Configuration Workshop*, p. 47, (2016).

[20] E. Teppan and G. Friedrich, 'Heuristic Constraint Answer Set Programming', in *ECAI 2016*, pp. 1692–1693, (2016).

[21] J. Tiihonen, T. Soininen, I. Niemelä, and R. Sulonen, 'A practical tool for mass-customizing configurable products', in *14th International Conference on Engineering Design*, pp. 1290–1299, (2003).

[22] E. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, London, 1993.

[23] A. Weinzierl, 'Blending lazy-grounding and CDNL search for answer-set solving', (2017).