ASCASS: A Simple Constraint Answer Set Solver - Technical Report

Erich C. Teppan and Gerhard Friedrich
Alpen-Adria Universitaet Klagenfurt
firstname.lastname@aau.at

Abstract

Constraint answer set programming (CASP) is a family of hybrid approaches integrating answer set programming (ASP) and constraint programming (CP). These hybrid approaches have already proven to be very successful in various domains. In this paper we present the CASP solver ASCASS (A Simple Constraint Answer Set Solver) which provides novel methods for defining and exploiting search heuristics. Beyond the possibility of using already built-in problem-independent heuristics, ASCASS allows on the ASP level the definition of problem-dependent variable selection, value selection and pruning strategies which guide the search of the CP solver. The concepts are exemplified and evaluated with respect to the real world Partner Units Problem (PUP). Due to a sophisticated heuristic, which cannot be represented by other ASP or CASP solvers, ASCASS shows superior performance.

1 Introduction

During the last decade, Answer Set Programming (ASP) under the stable model semantics [Gelfond and Lifschitz, 1988] has evolved to an extremely powerful approach for solving combinatorial problems. Especially conflict-driven search mechanisms contributed to the high performance of state-of-the-art solvers [Gebser *et al.*, 2012]. Furthermore, ASP provides superior problem encoding capabilities as ASP is strongly declarative in nature and even provides language features which go beyond first order.

However, the expressive power on the one hand and the mighty conflict-driven search approach on the other hand does not come for free. Current ASP solvers employing conflict-driven search transform the higher-order problem representation to propositional logic. This transformation (called grounding) constitutes the space bottleneck of nowadays ASP systems. Once the grounding step is completed, the performance of the conflict-driven search in combination with state-of-the-art look-back heuristics like VSIDS and restarts [Lewis *et al.*, 2005] typically shows superior performance compared to other search approaches. Yet, grounding is not possible for many industrial-sized problem instances.

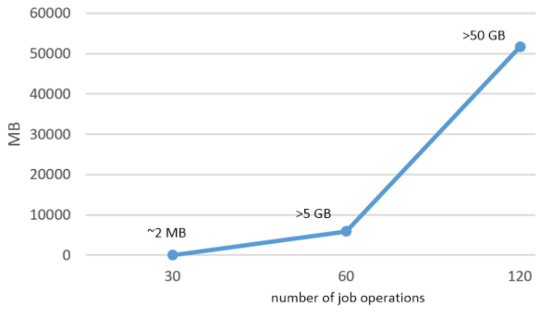


Figure 1: Grounding size for the incremental scheduling problem with respect to number of job operations

Firgure 1 shows how the size of the grounding explodes for the incremental scheduling problem instances from the ASP competition[1]. For the instances incorporating 120 job operations the size of the grounding is more than 50 GByte.

In industrial scheduling domains the sizes of problem instances are typically significantly higher. Scheduling instances of our project partners Infineon Austria incorporate $>$10000 job operations for a weekly workload performed on $>$100 machines in the back-end (i.e. where chips are cut and packaged) and $>$100000 job operations for a weekly workload performed on $>$1000 machines in the front-end (i.e. where the chips are actually produced). Thus, such instances are clearly out of reach for conventional ASP approaches.

One approach that emerged also out of the need of easing the grounding was Constraint Answer Set Programming (CASP) [Mellarkod *et al.*, 2008]. CASP can be seen as a hybrid approach extending ASP by Constraint Programming (CP) features. Conceptually, it is very close to satisfiability (SAT) modulo theory approaches which integrate first-order formulas with additional background theories such as real numbers or integers [Sebastiani, 2007].

For combining ASP and CP there are basically two approaches. First, solvers like Clingcon [Ostrowski and Schaub, 2012] are based on the extension of the ASP input language in order to support the definitions of constraints. A different approach has been introduced by [Balduccini, 2009] where ASP

[1]Find instances encodings and grounders/solvers at www.mat.unical.it/aspcomp2014. Our tests were done with gringo4 and the provided 'new' encoding.

and CP are not integrated into one language. ASP rather acts as a specification language for Constraint Satisfaction Problems (CSPs). The main idea is that answer sets constitute CSP encodings which are used as input for a CP solver.

For certain classes of problems like industrial-sized scheduling CASP was already successfully applied [Balduccini, 2011]. Especially search problems with large variable domains often profit from the CASP representation due to the alleviation of the grounding bottleneck [Lierler *et al.*, 2012].

Of course, the complexity of problem solving does not vanish by easing the grounding bottleneck but it is rather shifted from grounding in ASP to search in CP. In the context of CASP, often a majority of the solution calculation is done by the CP solver. Hence, the applied search strategies on the CP level play a crucial role for the successful application of CASP in real-world problem domains. However, up to now there was no focus on the development of sophisticated features for expressing and exploiting search strategies in CASP solvers. Consequently, the means for expressing and exploiting search strategies on the CP level are rather limited.

Clingcon does not provide any means for influencing the search of the underlying CP solver Gecode[2]. In EZCSP[3] (see [Balduccini, 2009]) there is a set of built-in strategies depending on the underlying CP solver that can be used. In case of Sicstus Prolog as a CP solver, the value selection strategies *step* (min domain value, when ascending order is used, max domain value when descending order is used) and *bisect* (bisection of the domain in the middle) are available. Similarly in case of B-Prolog, the bisection strategies *split* and *reverse_split* are supported. The supported variable selection strategies are *leftmost* (leftmost variable), *min* (leftmost variable with minimal lower bound), *max* (leftmost variable with maximal upper bound), and *ff* (first-fail).

Clearly, for many real-world problem domains problem-independent strategies are not sufficient and problem-dependent heuristics are needed. Any problem-dependent heuristic on the CP level basically consists of three components:

1. a problem-dependent variable selection strategy
2. a problem-dependent value selection strategy
3. a problem-dependent pruning strategy

In EZCSP, problem-dependent variable selection strategies are already possible. By the special predicate *label_order* it is possible to define the order in which the CSP variables are processed by the CP solver. What is missing is the possibility of expressing custom value ordering and pruning strategies.

In this paper we present ASCASS, a novel CASP solver which uses Clingo for answer set solving and the Java framework Jacop for CP solving. ASCASS combines and extends the heuristic possibilities of state-of-the-art CASP solvers and makes them completely available on the problem encoding level. Beyond the usage of built-in strategies, ASCASS provides powerful constructs for the formulation and exploitation of problem-dependent heuristics consisting of variable selection, value selection and pruning strategies.

By means of the real-world Partner Units Problem (PUP), which constitutes one of the hardest benchmarks in the ASP competition, we exemplify problem encoding in ASCASS. We furthermore show how to express the currently most powerful heuristic for this problem in ASCASS. It is shown that due to this heuristic, which, to the best of our knowledge, can not be expressed within any other ASP or CASP approach, ASCASS outperforms state-of-the-art ASP and CASP solvers.

2 A Simple Constraint Answer Set Solver

Architecture ASCASS[4] is a CASP solver following the approach of [Balduccini, 2009], i.e. the input language is pure ASP and the answer sets encode CSPs. Figure 2 shows the overall architecture of ASCASS. Answer set production (grounding and solving) is done by Clingo[5], which is currently one of the most powerful ASP systems. The input language is the ASP standard ASP-Core-2[6].

After answer set solving, a produced answer set is handed over to a parsing module that extracts the facts which encode the CSP and search directives. This information is used to instantiate a corresponding CSP in the CP solver and perform search conforming to the given search directives. Currently, Jacop[7] is used within ASCASS as a CP solver. In case that the CSP could not be solved by the CP solver or a timeout occurred (defined by the special predicate $csptimeout(\Delta)$), the process continues with the next answer set, until a solution is found, or there are no more answer sets. The empty CSP (i.e. when there is not a single CSP variable) is always satisfiable and possesses the empty CSP solution.

Encoding of CSPs ASCASS focuses on finite discrete Constraint satisfaction problems (CSPs). CSPs can be defined as three-tuples of the form $\langle V, D = \{dom(v) | v \in V\}, C\rangle$ whereby V is a set of variables, D is the set of domains of the variables in V and C is a set of constraints on variables in V. A solution to a CSP is an assignment $\forall v \in V, v := d \in dom(v)$ such that all constraints $c \in C$ are fulfilled. In order to encode a CSP within ASCASS there can be used a number of specific predicates. Of course, in the input these predicates can contain variables. The following explanations refer to their grounded form.

The predicates $cspvar(\alpha, \lambda, \upsilon)$ and $cspvar(\alpha, \lambda, \upsilon, \eta)$ are responsible for encoding CSP variables. Hereby, α represents the variable name and λ and υ represent respectively the numerical lower and upper bound of the variable's domain. For example $cspvar(x, 1, 10)$ stands for a CSP variable v with the domain $[1..10]$. The numerical priority η is used to define a custom variable selection ordering. When using the variable selection strategy $priority$ (see below), the CP solver selects the variable with the highest priority first.

The predicate $cspconstr(\alpha, \rho, \tau)$ encodes a relational constraint (i.e. $=, <>, <, <=, >, >=$) over a variable α. ρ

[2]www.gecode.org

[3]mbal.tk/ezcsp/index.html

[4]www.dropbox.com/l/sh/A0jQqrE9kiNTXDmV9FrQhr

[5]sourceforge.net/projects/potassco/files/clingo

[6]www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03b.pdf

[7]jacop.osolpro.com

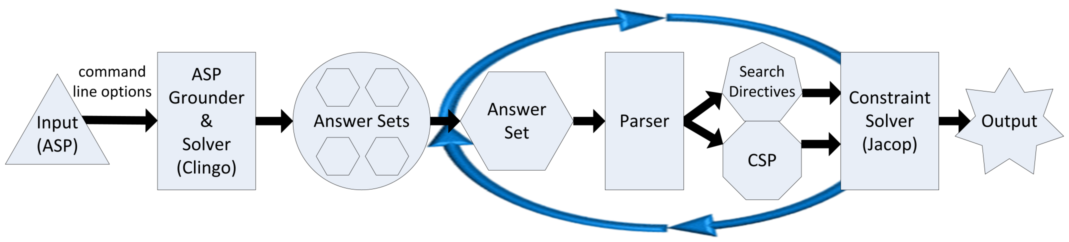


Figure 2: Architecture of ASCASS

denotes the type of relation and must be a constant out of $\{eq, neq, lt, lteq, gt, gteq\}$. τ denotes another CSP variable or a numerical constant. For example, $cspconstr(x, lt, 5)$ expresses that variable x must be lower than 5.

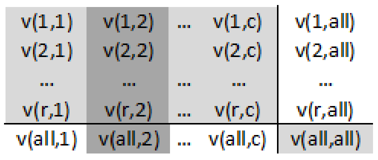
The predicate $csparith(\alpha, \pi, \beta, \rho, \gamma)$ encodes arithmetic constraints. α, β and γ are CSP variable names. Like for $cspconstr$, the constant ρ denotes the type of relation. π is a constant standing for an arithmetic operation. Currently, ASCASS supports addition ($plus$), subtraction ($minus$), multiplication ($mult$), division (div) and exponent (exp). For example, $csparith(xa, plus, xb, eq, xc)$ states that the sum of the values of xa and xb must be equal the value of xc.

For expressing logical constraints predicates of the form $cspif(\Xi_1, and, \Xi_2, and, \ldots, and, \Xi_m, then, \Xi_{m+1}, or, \Xi_n)$ can be used. Each Ξ consists of a variable α, a relational symbol ρ and another variable or numerical constant τ. For example, $cspif(x, lt, 5, and, y, gt, 10, then, z, gteq, 0)$ is to be read as 'if x is lower than 5 and y is greater than 10 then z must be non-negative'.

Global constraints are constraints over arrays of variables. In ASCASS global constraints are defined by predicates of the form $cspglobal(\sigma_1, \ldots, \sigma_m, \kappa)$ and $cspglobal(\sigma_1, \ldots, \sigma_m, \kappa, \tau_1, \ldots, \tau_n)$. κ is a constant denoting the type of global constraint. $\sigma_1, \ldots, \sigma_m$ represent arrays of variables. $\tau_1, \ldots, \tau_n$ represent single CSP variables or integers. ASCASS currently supports the following global constraints[8]:

- min: $cspglobal(\sigma, min, \tau)$, the minimum value of the variables σ is equal to τ
- max: $cspglobal(\sigma, max, \tau)$, the maximum value of the variables σ is equal to τ
- sum: $cspglobal(\sigma, sum, \tau)$, the sum of values of the variables σ is equal to τ
- count: $cspglobal(\sigma, count, \tau_1, \tau_2)$, τ_1 is equal to the counted number of variables in σ with value τ_2
- global cardinality: $cspglobal(\sigma_1, \sigma_2, gcc)$, a more general counting constraint where the occurring values in σ_1 are counted in the corresponding counter variables in σ_2
- all different: $cspglobal(\sigma, alldiff)$, all variables in σ are mutually unequal

[8]More information about global constraints can be found at http://jacop.osolpro.com/guideJaCoP.pdf and http://sofdem.github.io/gccat/



v(1,1)	v(1,2)	...	v(1,c)	v(1,all)
v(2,1)	v(2,2)		v(2,c)	v(2,all)
...	...	...	...	...
v(r,1)	v(r,2)	...	v(r,c)	v(r,all)
v(all,1)	v(all,2)	...	v(all,c)	v(all,all)

Figure 3: Concept of variable arrays in ASCASS

- element: $cspglobal(\sigma, element, \tau_1, \tau_2)$, the value of the τ_1-th variable in σ is equal to τ_2
- cumulative: $cspglobal(\sigma_1, \sigma_2, \sigma_3, cumulative, \tau)$, σ_1 represents the starting times of $|\sigma_1|$ many jobs, σ_2 represents the durations of the jobs, σ_3 represents the amounts of needed resources of the jobs and τ represents the allowed accumulated amount of resources at any time point
- bin packing: $cspglobal(\sigma_1, \sigma_2, \sigma_3, binpacking)$, σ_1 represents bin assigments for $|\sigma_1|$ many items, σ_2 represents the bin sizes of the $|\sigma_2|$ many bins and σ_3 represents the item sizes

In order to address arrays of CSP variables, ASCASS not only allows simple constants but also n-ary functional terms for variable names of the form $\phi(\iota_1, \ldots, \iota_n)$ with $\iota_1, \ldots, \iota_n$ representing string or integer arguments (see Figure 3). The special functional argument *all* acts as a placeholder and can be used for addressing arrays of variables. For example, take the four variable definitions $cspvar(v(1, 1), 1, 10)$, $cspvar(v(1, 2), 1, 10)$, $cspvar(v(2, 1), 1, 10)$ and $cspvar(v(2, 2), 1, 10)$. A natural interpretation of the arguments is *row* and *column* of a two-dimensional variable array. Consequently, $cspglobal(v(all, 2), alldiff)$ expresses that the values of all second column's variables, in our case $v(1, 2)$ and $v(2, 2)$, must be different to each other. *v(all,all)* stands for all variables in the two-dimensional array, i.e. all variables formed by the functional symbol v with arity 2.

Encoding of variable selection strategies Apart from the predicates for defining a CSP, ASCASS provides predicates for steering the search of the CP solver. The predicates $cspvarsel(\epsilon)$ and $cspvarsel(\epsilon, \theta)$ define the variable selection strategy to be used. Herby, ϵ is the primary selection strategy and, if defined, θ acts as a tiebreaker.

For variable selection, ASCASS currently supports the problem-independent built-in strategies *smallestDomain*, *mostConstrainedStatic*, *mostConstrainedDynamic*, *smallestMin*, *largestDomain*, *largestMin*, *smallestMax*, *maxRegret*, *weightedDegree* and the problem-dependent strategy *priority*.

When using the *priority*-strategy, ASCASS builds an ordering of the CSP variables based on the provided priorities η in $cspvar(\alpha, \lambda, v, \eta)$. Variables with high priorities are selected first. Variables for which there is no η defined are selected as the last ones. Hence, the $priority$ strategy in combination with the variable priorities is similar to the *label_order* predicate in Balduccini's EZCSP.

Encoding of value selection strategies For value selection ASCASS provides the predicates $cspvalsel(\phi)$ and $cspvalsel(\phi, \varphi)$ where ϕ and φ are constants denoting the strategy. As it is often important to have different value selection strategies for different sets of variables, ASCASS provides also the predicates $cspvalsel(\sigma, \phi)$ and $cspvalsel(\sigma, \phi, \varphi)$ where σ represents an array of variables like in global constraints. ASCASS supports the already built-in strategies *indomainMin*, *indomainMiddle*, *indomainMax* and *indomainRandom*. For expressing problem-dependent value selection strategies, the novel strategy *indomainPreferred* can be used.

When using *indomainPreferred*, the CP solver first tries to use specified values before changing to the built-in strategy φ (*minDomain* if not stated otherwise). For specifying preferred values, ASCASS provides the special predicate $cspprefer(\alpha, \rho, \tau)$ and $cspprefer(\alpha, \rho, \tau, \eta)$. Like for relational constraints, α represents a CSP variable, ρ represents a relational symbol and τ stands for a further variable or a numerical constant. For example, $cspprefer(v, eq, 5)$ states that for the CSP variable v a preferred value is 5. In order to specify an ordering of the specified values, it is possible to make use of a numerical priority η. Higher priority statements are taken into account first by ASCASS. For example, if there is given $cspprefer(v, eq, 5, 1)$ and $cspprefer(v, eq, 20, 2)$, ASCASS tries to first label v with 20 and only after that with 5. Of course, only preferred values are taken into account which are still in the variable's domain. In case that τ denotes another variable, the minimum value in the current domain of τ is used as a preferred value, i.e. τ does not need to be singleton for specifying a preferred value of α. This in combination with global constraints is a highly dynamic and powerful mechanism.

As with the relational constant eq in combination with the priorities η every ordering of preferred values can be expressed, the usage of lt, $lteq$, gt and $gteq$ can be clearly seen as syntactic sugar. By using lt, $lteq$, gt and $gteq$ sets of preferred values can be expressed:

- $lteq\,\tau : \{\tau, \tau-1, \ldots, -\infty\}$
- $lt\,\tau : \{\tau-1, \ldots, -\infty\}$
- $gteq\,\tau : \{\tau, \tau+1, \ldots, \infty\}$
- $gt\,\tau : \{\tau+1, \ldots, \infty\}$

Note that all preferred values of such a set P have the same priority (possibly given explicitly by η). For defining an order relation over P, i.e. fix the order in which ASCASS considers the preferred values in P, the following holds: For lt and $lteq$ decreasing order is used, i.e. $\tau, \tau-1, \ldots, -\infty$ and for gt and $gteq$ increasing order is used, i.e. $\tau, \tau+1, \ldots, \infty$. For example having the variable definition $cspvar(v, 1, 10)$ and the value selection strategy $cspvalsel(indomainPreferred, indomainMin)$, $cspprefer(v, lt, 5)$ would effect that ASCASS considers the domain values in the following order: $4, 3, 2, 1, 5, 6, 7, 8, 9, 10$. The reason why for lt and $lteq$ descending order and for gt or $gteq$ ascending order is used is simply the following: Would it be the other way round, the behavior with lt and $lteq$ would conform to $indomainMin$ and with gt and $gteq$ to $indomainMax$.

Encoding of pruning strategies The third component of many problem-dependent heuristics is the pruning strategy. For specifying how a search tree is pruned, ASCASS provides the special predicate $cspsearch(\omega, \mu)$. Hereby, ω specifies the pruning type and μ specifies a numerical limit that, when reached, triggers backtracking. Again it could be beneficial having different limits for different groups of variables or even having no limit on certain variables whilst search on others is limited. To this, ASCASS provides the predicate $cspsearch(\sigma, \omega, \mu)$ with σ denoting an array of variables like for global constraints.

Currently, ASCASS provides two pruning types. $cspsearch(limit, \mu)$ limits the number of wrong decisions for variables. If the number μ of wrong choices for a variable is reached, backtracking is triggered and the counter for the variable is reset. For example, $cspsearch(limited, 3)$ specifies that for every variable v there must not be more than three labeling trials for v within a search branch. The second pruning type is based on limited discrepancy search [Harvey and Ginsberg, 1995] and operates on the level of search paths. When specifying $cspsearch(lds, \mu)$ only a certain number of wrong decisions (called discrepancies) along the whole search path is allowed. If this number reaches μ, backtracking is triggered.

3 Proof of Concept

We want to exemplify the expressive power of ASCASS with respect to the partner units problem (PUP) out of three reasons.

1. The PUP is a real world combinatorial problem with many different application domains [Aschinger *et al.*, 2011].
2. The PUP is one of the hardest benchmark problems participating in the ASP competitions[9].
3. There exists an effective problem-dependent heuristic to solve the PUP.

The PUP originates in the domain of railway safety systems. One of the problems in this domain is to make sure that certain rail tracks are not occupied by a train/wagon before

[9]Further information can be found at www.mat.unical.it/aspcomp2014/

another train enters this track. The signals for the corresponding occupancy indicators are calculated by special processing units based on the input of several observing sensors. Because of fail-safety and realtime requirements the number of sensors respectively indicators which can be connected to the same unit is limited (called unit capacity, UCAP). Also one sensor/indicator device can only be directly connected to one unit. However, a unit can be connected to a limited number (called inter unit capacity, IUCAP) of other units. These units are called the partner units of the unit. Devices (i.e. sensors and indicators) can only communicate with devices connected to the same unit and with devices connected to one of the partner units. Given the IUCAP, UCAP and a bipartite input graph represented by edges specifying which sensor data is needed in order to calculate the correct signal of an occupancy indicator, the problem consists in connecting sensors/indicators with units and units with other units such that all communication requirements are fulfilled and IUCAP and UCAP are not violated.

The state-of-the-art heuristic for solving PUP is the QuickPup heuristic proposed in [Teppan *et al.*, 2012]. QuickPup is based on three major techniques. First, based on the input graph and a distinguished root indicator, QuickPup produces a topological ordering of the devices, which is basically the minimum distances from the root indicator to all other devices. The distance to itself is zero, the distance to the direct neighbors is one, the distance to the neighbors of the neighbors is two and so forth. This reflects the (partial) ordering in which the devices should be processed. Second, for each device, first try to place it on the next empty unit and if this is unsuccessful try the already used units in descending order. Third, try different root indicators, and consequently different topological orderings, and limit search for each trial. The intuition behind that is that not all root indicators are equally good to start search from.

The input comprises of a set of $egde(i, s)$ facts where i takes the numerical id of an indicator and s takes the id of a sensor. Additionally the input includes a fact $ucap(x)$ with $x > 0$ that defines the unit capacity (UCAP) and a fact $iucap(y)$ with $y > 0$ that defines the inter-unit capacity (IUCAP).

For the code snippets given in the remainder of this section we use the standard notation of logic programming and explain extensions regarding ASP as needed. For the purposes of this paper, answer sets can be seen as minimal logic models and answer set production can be thought of applying forward chaining.

In order to produce explicit indicator and sensor information the following lines of code are used:

```
sensor(S):-edge(I,S).
indicator(I):-edge(I,S).
numIndicators(N):-N=#count{I:indicator(I)}.
numSensors(N):-N=#count{S:sensor(S)}.
```

The number of indicators ($numIndicators$) respectively sensors ($numSensors$) are calculated by means of the $\#count$ aggregate literal provided by Clingo.

We restrict the number of units ($numUnits$) available for a solution to the theoretical minimum plus two, i.e. $numUnits = \left\lceil \frac{max(numIndicators, numSensors)}{UCAP} \right\rceil + 2$:

```
max(M):-numIndicators(E),numSensors(F),M=#max(E;F).
numUnits(N):-max(M),ucap(C),N=((M+1)/C)+2.
unit(Z):-numUnits(N),1<=Z,Z<=N.
```

Of course, the number of units is to be changed for optimization purposes.

For each indicator i there is a CSP variable $device(i, 1)$ and for each sensor s there is a CSP variable $device(s, 2)$. This way it is also possible to refer to the array of all CSP device variables as $device(all, all)$, to only the indicator variables as $device(all, 1)$ and to the sensor variables as $device(all, 2)$ which will be useful later. The value range for these CSP variables is $[1..numUnits]$. Furthermore, the variables get a priority defining the topological order in which they are labeled by ASCASS:

```
cspvar(device(I,1),1,N,P):-numUnits(N),iPriority(I,P).
cspvar(device(S,2),1,N,P):-numUnits(N),sPriority(S,P).
```

The calculation of the priorities is explained in detail below.

In order to assure UCAP, for each unit u there are two counting variables $ci(u)$ and $cs(u)$. These variables can take values in the range $[0..UCAP]$. Furthermore, for each unit u there are two $count$ global constraints counting the number of indicator respectively sensor variables taking the value u:

```
cspvar(ci(U),0,C):-ucap(C),unit(U).
cspvar(cs(U),0,C):-ucap(C),unit(U).
cspglobal(device(all,1),count,ci(U),U):-unit(U).
cspglobal(device(all,2),count,cs(U),U):-unit(U).
```

In order to capture which unit $u1$ is connected to which unit $u2$ there are $numUnits \times numUnits$ many CSP variables (i.e. $conn(U1, U2)$). The variables can take values in the range $[0..1]$ if $u1 <> u2$. Otherwise, the variables' ranges consists of only a single value, i.e. $[1..1]$. This is because in our model each unit u is always connected to itself. Furthermore, there is a constraint assuring symmetry, i.e. if $u1$ is connected to $u2$ also $u2$ is connected to $u1$:

```
cspvar(conn(U1,U2),0,1):-unit(U1),unit(U2),U1<>U2.
cspvar(conn(U,U),1,1):-unit(U).
cspconstr(conn(U1,U2),eq,conn(U2,U1)):-unit(U1),unit(U2),
                                                     U1<U2.
```

For summing up how many units are connected to a unit u we make use of the global sum constraint. The used summing variables can hereby take values in the range $[1..IUCAP + 1]$ as every unit is also connected to itself:

```
cspvar(sumconns(U),1,K+1):-iucap(K),unit(U).
cspglobal(conn(U,all),sum,sumconns(U)):-unit(U).
```

In order to make the summing variables and constraints take effect, it must be assured that any connection variable $conn(u1, u2)$ is set to one whenever there is an $edge(i, s)$ in the input so that $device(i, 1) = u1$ and $device(s, 2) = u2$. Following the approach of [Drescher, 2012], this is implemented by means of the global $element$ constraint. Given an array of CSP variables arr, an index i and a value v, an $element$ constraint assures that the i^{th} variable in arr is equal to v. In our case, for each $edge(i, s)$ in the input there is such a global constraint setting the appropriate connection variable within $conn(all, all)$ to one:

```
cspglobal(conn(all,all),element,index(I,S),1):- edge(I,S).
```

As the $element$ constraint cannot directly handle multi-dimensional arrays, the respective index is calculated as $index(i, s) = (device(i, 1)-1) \times numUnits + device(s, 2)$.

The priorities for the device variables (i.e. $device(i, 1)$ and $device(s, 2)$ are based on a topological ordering of the devices. Given the layer of a sensor or indicator whereby the root of the topological graph is at layer zero, the priority is higher the lower the layer is:

```
iPriority(I,P):-indicatorLayer(I,L),P=9999-L.
sPriority(S,P):-sensorLayer(S,L),P=9999-L.
```

The effect is that given a root indicator, ASCASS first tries to label the root indicator, then the neighbors of the root indicator, then the neighbors of the neighbors, and so on. In our implementation a choice rule is used to express that there is exactly one distinguished indicator that acts as root. This indicator is always placed at the first unit:

```
1{root(I):indicator(I)}1.
cspconstr(device(I,1),eq,1):-root(I).
```

The choice rule $1\{root(I) : indicator(I)\}1$ produces one answer set for each root indicator and asserts a $root(i)$ fact.

For calculating the actual layers, we first calculate the minimum distances to the root whereas root indicator has a zero distance to itself[10]:

```
indicatorDist(I0,0):-root(I0).
sensorDist(S,D+1):-indicatorDist(I,D),edge(I,S),
                                               numDevices(M),D<M.
indicatorDist(I,D+1):-sensorDist(S,D),edge(I,S),
                                               numDevices(M),D<M.
numDevices(N):-numIndicators(E),numSensors(F),N=E+F.
```

The layers are calculated by using the $\#min$ aggregate literal from Clingo:

```
indicatorLayer(I,Dmin):-indicator(I),
                          Dmin=#min{D:indicatorDist(I,D)}.
sensorLayer(S,Dmin):- sensor(S),
                          Dmin = #min{D:sensorDist(S,D)}.
```

First to try to place devices on unused units and, only if not successful, on used units in descending order can be expressed in ASCASS by means of preferred values:

```
cspprefer(device(I,1),lteq,nextUnit):-indicator(I).
cspprefer(device(S,2),lteq,nextUnit):-sensor(S).
```

The CSP variable $nextUnit$ points to the next unused unit, which is the last used unit plus one[11]:

```
cspvar(lastUnit,1,N):-numUnits(N).
cspvar(nextUnit,1,N+1):-numUnits(N).
csparith(lastUnit,plus,one,eq,nextUnit).
```

For the calculation of the last used unit, i.e. the highest number taken by some $device(i, 1)$ or $device(s, 2)$ variable, the global max constraint is used:

```
cspglobal(device(all,all),max,lastUnit).
```

As ASCASS uses the lower bound of variables for calculating the preferred values, each device variable is first tried to be bound to values lower than or equal to the lower bound of $nextUnit = lastUnit + 1$ in descending order.

In order to control how many units are maximally tried per device variable, the search is pruned such that only the next unit and a limited number of already used units can be tried before backtracking is triggered. In our implementation we use the following statement for only trying the next and the last unit:

```
cspsearch(limited,2).
```

For making ASCASS respect the problem-dependent selection strategies, $cspvarsel(priority)$ and $cspvalsel(device(all, all), indomainPreferred)$ must be included. Thus, QuickPup can be fully expressed in a declarative way by ASCASS. To the best of our knowledge, this is not possible within any other ASP or CASP approach.

[10]In order to make grounding safe, we have to limit the maximum possible distance which is equal to the total number of devices.

[11]Within the constraint, the helping variable $cspvar(one, 1, 1)$ is used as arithmetic constraints only accept variables in ASCASS.

	#	**Clingo**	**ASCASS**	**Clingcon**	**Ezcsp**
double(IUCAP=2)	10	2	10	0	2
doublev(IUCAP=2)	6	3	6	0	0
triple(IUCAP=2)	5	2	5	0	2
triple(IUCAP=4)	7	6	7	0	3
grid(IUCAP=4)	10	10	10	0	0
total	38	23	38	0	7

Table 1: Solved instances whithin 600 seconds

Evaluation We tested the ASP solver Clingo 4 and the CASP solvers ASCASS, Clingcon and Ezcsp on the PUP benchmark suite used in [Aschinger *et al.*, 2011][12]. Clingo was tested using the PUP encoding proposed in [Aschinger *et al.*, 2011]. The tests were run on a 3.2 Ghz machine with 64 GByte of RAM, assuring that the grounding bottleneck does not play a role for the tested instances[13] and performance can be attributed to the search phase.

In the Clingcon model, CSP variable selection, value selection or pruning strategies cannot be manipulated. For Ezcsp, it is possible to express the topological variable orderings similar to ASCASS. However, there are no means for pruning search or problem-dependent value strategies.

Table 1 depicts how many instances of each type in the benchmark suite could be solved by the different approaches within a 600 seconds time frame. Clingo using VSIDS heuristic peformed very well on the benchmark suite showing once again that the conflict-driven search techniques employed by Clingo are quite powerful. Also Ezcsp was able to solve some instances. Using other built-in heuristics did not result in better performance. Clingcon was not able to solve a single instance. This supports the finding in [Lierler *et al.*, 2012] that propagation between ASP and CP does not yet perform optimal in Clingcon. In the contrary, ASCASS was able to solve all 38 instances before a timeout occurred. In fact, every instance could be solved within (typically much) less than 70 seconds which can be attributed to the sophisticated QuickPup heuristic. This was crosschecked by removing the heuristic from the ASCASS problem encoding which effected that also no instance could be solved within time limits.

Conclusions It can be said that constraint answer set programming (CASP) tries to combine the best from two different worlds. Within this scope, ASCASS provides superior features for using search heuristics. Apart from built-in problem-independent heuristics, ASCASS facilitates the exploitation of problem-dependent variable selection, value selection and pruning strategies. On the back of the real-world Partner Units Problem, which constitutes one of the hardest benchmark problems of the ASP competitions, we exemplified problem encoding in ASCASS. We could show that the non-trivial problem-dependent QuickPup heuristic can be expressed quite naturally in ASCASS. Due to the heuristic, which cannot be expressed by any other ASP or CASP system, ASCASS outperforms state-of-the-art ASP or CASP systems on the tested instances.

[12]www.dropbox.com/l/sh/A0jQqrE9kiNTXDmV9FrQhr

[13]The biggest grounding in the ASP model was $\sim$ 12 GByte.

Acknowledgements Work has been funded by the Austrian Research Fund (FFG) in the context of project *HINT* (PNr.: 840242).

References

[Aschinger *et al.*, 2011] M. Aschinger, C. Drescher, G. Friedrich, G. Gottlob, P. Jeavons, A. Ryabokon, and E. Thorstensen. Optimization methods for the partner units problem. In *CPAIOR'11*, pages 4–19, Berlin, Heidelberg, 2011. Springer-Verlag.

[Balduccini, 2009] M. Balduccini. Representing constraint satisfaction problems in answer set programming. In *ICLP09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'09)*, 2009.

[Balduccini, 2011] Marcello Balduccini. Industrial-size scheduling with asp+cp. In *Proceedings of the 11th Int. Conf. on Logic Programming and Nonmonotonic Reasoning*, LPNMR'11, pages 284–296, Berlin, Heidelberg, 2011. Springer-Verlag.

[Drescher, 2012] Conrad Drescher. The partner units problem: A constraint programming case study. In *ICTAI'12*, 2012.

[Gebser *et al.*, 2012] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth Int. Conf. and Symp. of Logic Progr. (ICLP'88)*, pages 1070 – 1080. MIT Press, 1988.

[Harvey and Ginsberg, 1995] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the 13th International Joint Conf. on Artificial Intelligence*, pages 607–613. Morgan Kaufmann, 1995.

[Lewis *et al.*, 2005] Matthew D. T. Lewis, Tobias Schubert, and Bernd W. Becker. Speedup techniques utilized in modern sat solvers. In *Proceedings of the 8th Int. Conf. on Theory and Applications of Satisfiability Testing*, SAT'05, pages 437–443, Berlin, Heidelberg, 2005. Springer-Verlag.

[Lierler *et al.*, 2012] Yuliya Lierler, Shaden Smith, Miroslaw Truszczynski, and Alex Westlund. Weighted-sequence problem: Asp vs casp and declarative vs problem-oriented solving. In *Proceedings of the 14th Int. Conf. on Practical Aspects of Declarative Languages*, PADL'12, pages 63–77, Berlin, Heidelberg, 2012. Springer-Verlag.

[Mellarkod *et al.*, 2008] Veena S. Mellarkod, Michael Gelfond, and Yuanlin Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 53(1-4):251–287, August 2008.

[Ostrowski and Schaub, 2012] Max Ostrowski and Torsten Schaub. Asp modulo csp: The clingcon system. *Theory Pract. Log. Program.*, 12(4-5):485–503, September 2012.

[Sebastiani, 2007] Roberto Sebastiani. Lazy satisfiability modulo theories. *J. on Satisfiability, Boolean Modeling and Computation*, 3:141–224, 2007.

[Teppan *et al.*, 2012] Erich Christian Teppan, Gerhard Friedrich, and Andreas Falkner. Quickpup: A heuristic backtracking algorithm for the partner units configuration problem. In *International Conference on Innovative Applications of AI (IAAI'12)*, pages 2329–2334. AAAI, 2012.