Solving B Constraints with Goal-directed Answer Set Programming

Alexandros Efremidis

Institut für Informatik, Heinrich-Heine-Universität Düsseldorf
Universitätsstraße 1, 40225 Düsseldorf, Germany
alefr101@hhu.de

Abstract. In this paper I explore a further option for solving B constraints. In particular, I develop a framework translating B predicates to s(CASP), a goal-directed form of Answer Set Programming. Furthermore, the presented framework implements an interface enabling B predicates to be solved by the s(CASP) engine within the ProB tool as an additional backend. This paper particularly focuses on the translation process and on empirically evaluating the framework's performance by comparing it to the native, Kodkod and Z3 backend of ProB. This work poses a foundation for future development regarding the translation of B predicates to goal-directed Answer Set Programming and s(CASP) specifically. Further, this framework can be used to aid the verification of other solvers' correctness.

1 Introduction and Motivation

This work's fundamental motivation is to develop zero-defect software with the help of formal methods. For instance, this is achievable by specifying a system's desired behavior with B abstract machines [1] to then be validated by a software verification tool such as ProB [7,8]. ProB is an automated analysis toolkit for the B-method enabling for animation, model checking and constraint solving, which allows for unveiling possible errors of the underlying specification. In order to model check a machine and verify its correctness, predicates are usually evaluated along the way. Therefore, constraint solvers such as the native ProB, Kodkod [13] and Z3 [5] backend are employed to support the overall verification process. As they all come with their respective strengths and weaknesses [6,11,12] it seems natural to explore further options.

In this paper I aim towards extending the existing base of constraint solvers for the B-Method, used in the ProB tool, by providing an additional constraint solving backend. In particular, this work implements a Prolog framework translating B predicates to Answer Set Programming [9] to then be solved by s(CASP) [2] within ProB. Furthermore, this paper focuses on the translation process and on empirically evaluating the framework's performance by comparing it to the aforementioned backends of ProB.

2 Answer Set Programming and s(CASP)

Answer Set Programming (ASP) is a form of declarative logic programming, which is primarily oriented towards NP-hard search problems [9]. Due to its declarative nature ASP poses an attractive alternative to already well-established constraint solving oriented methodologies and is a paradigm of growing interest in the recent years. An ASP program P is a finite set of clauses, where each rule $r \in P$ is of the form

$$h \leftarrow t_1 \wedge \cdots \wedge t_m \wedge not\ t_{m+1} \wedge \cdots \wedge not\ t_n \tag{1}$$

with the head h and the body's literals $t_1, \ldots, t_n$ being compound terms. The keyword *not* expresses default negation. Generally, ASP is concerned with finding stable models also referred to as answer sets for the underlying problem. In typical ASP applications an initial grounding phase is introduced in order to be able to search for answer sets. The grounding phase succeeds only if every clause of P is safe. A rule is considered as safe in case every variable in its body occurs in some positive literal thereby specifying the variable's finite domain. A rule is deemed unsafe otherwise. However, for this work an ASP implementation is chosen, which is free of grounding.

s(CASP) [2] is a novel Answer Set Programming implementation coalescing stable model semantics and Constraint Logic Programming (CLP). It is build upon the s(ASP) [10] execution model, an ASP interpreter written in Prolog. s(CASP) inherits and generalizes s(ASP) while remaining parametric with respect to CLP. In particular, s(CASP) is a query-driven implementation of Answer Set Programming. Unlike common ASP systems, s(CASP) does not employ any SAT based methodology and is not relying on a grounding phase either prior or during execution. Hence, s(CASP) allows for declaring variables without specifying their respective domains, whereas classical ASP approaches would consider the program as unsafe. s(CASP) deploys a top-down, query-driven procedure virtually an extended version of SLDNF resolution for evaluating programs under the ASP semantics. By virtue of s(CASP) being goal-directed the engine computes a partial stable model, which is the portion required for answering the underlying query.

3 A Constraint Solving Framework

This framework consists of a translator and an interface. The translator obtains an abstract syntax tree (AST) representing a B predicate emitted by ProB and generates semantically equivalent s(CASP) code. It is written in SICStus Prolog [4] to ensure compatibility and is constructed as a loadable extension package of ProB. The interface establishes the connection between the translator and the s(CASP) engine. s(CASP) is written in Ciao Prolog [3], hence an interface links the two Prolog implementations together enabling for solving a B predicate by s(CASP) within ProB.

The overall workflow is illustrated in Figure 1. An AST is expected as input by the framework, hence ProB parses the predicate prior to the translation. This initial process is depicted as a black box in the flowchart. Thereafter, the translator generates a s(CASP) file containing executable code. The translator gradually walks over the AST and computes for each encountered B node a semantically equivalent s(CASP) component. Through this process a s(CASP) program is recursively build. When the generation of the translation is completed, the interface calls the s(CASP) engine and obtains the result. Lastly, some post-processing is applied and the result is returned back to ProB.

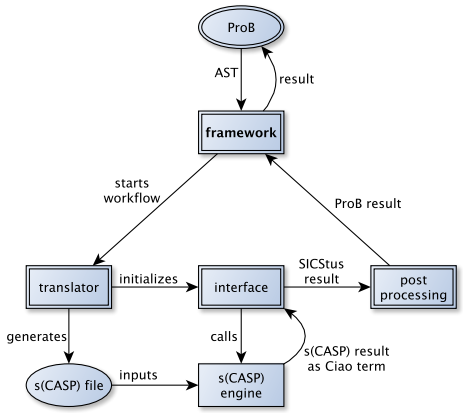

Fig. 1: The framework's general workflow.

4 Translating B Predicates to s(CASP)

In this section I give an overview of the translation process. The following designations are used for B code throughout this translation. Designations `P`, `Q` denote predicates; `E`, `F` denote expressions; `x`, `y` denote single variables; `z` denotes a list of variables; `S`, `D` denote set expressions; `U` denotes a set of sets and `m`, `n` denote integer expressions. Furthermore, for any B code `A`, $\texttt{T}_{\texttt{A}}$ denotes the translation of `A` in s(CASP). For the sake of simplicity the designations of B identifiers are also used for their s(CASP) representatives. For a B expression `E` the variable $\texttt{V}_{\texttt{E}}$ is unified with the evaluation of $\texttt{T}_{\texttt{E}}$. Lastly, `Tmp` denotes a fresh internal temporary variable.

The vast majority of B predicates is covered. However, a few restrictions are introduced as this framework is still under development. Predicates such as set summation, set product, iteration, closure, projection, lambda abstraction and the support of sequences are not incorporated. Furthermore, infinite domains of the form $a \in \mathbb{N}$ or $b \in \mathbb{Z}$ are not considered. Lastly, in pure logic the order of `P` and `Q` is irrelevant for the semantics of a conjunction. However, conjunctions are

currently straightforwardly translated. Therefore, the case that a ground value is needed for evaluating `T_P`, which remains unbound until `T_Q` is evaluated, is generally not supported. Nevertheless, s(CASP)'s CLP library allows for supporting arithmetical expressions.

Primitive expressions: Identifiers, booleans, integers and strings are straightforwardly translated.
Conjunction: `P & Q` is translated to `T_P, T_Q`.
Disjunction: `P or Q` introduces a goal `subconst(x_1,...,x_k)` with `x_1,...,x_k` being the variables occurring in `P` and `Q`. For both predicates a new rule is added to the code with the aforementioned goal as its head and with `T_P`, `T_Q` as its body respectively. Hence, the new goal establishes a choice point, effectively creating a disjunction.
Negation: Let `x_1,...,x_k` be the variables occurring in `P`. `not P` introduces the goal `not subconst(x_1,...,x_k)` with the rule of the form
`subconst(x_1,...,x_k) :- T_P`.
Implication: `P => Q` is resolved as `not P or Q`.
Equivalence: `P <=> Q` is resolved as `P => Q & Q => P`.
Existential quantification: By virtue of s(CASP) operating query-driven, `#(z).(P & Q)` can be resolved as `P & Q`.
Equality, inequality and comparison operators: This translation applies to the operators $=, \neq, <, >, \leq, \geq$. Without loss of generality, equality is selected to describe how the aforementioned operators are translated. As `E = F` may contain nested expressions or function calls the translator analyzes the AST to gather information about `E` and `F`. The translator operates with a look-ahead of one and introduces an internal temporary variable `V_e` if necessary. Depending on `E` and `F` the framework distinguishes between the following cases.

1. If `E` and `F` are primitive: `T_E = T_F`.
2. If `E` is primitive and `F` is not: `T_F, T_E = V_F`.
3. If `E` is non-primitive and `F` is primitive: `T_E, V_E = T_F`.
4. If `E` and `F` are both non-primitive: `T_E, T_F, V_E = V_F`.

Furthermore, the operators $\neq, <, >, \leq, \geq$ are translated to `\=, #<, #>, #=<, #>=` respectively. Thus, in case of arithmetical expressions s(CASP)'s CLP backend takes care of the aforementioned temporal challenges related to conjunctions.
Empty set: Sets are represented by lists, hence {} is translated to `[]`.
Singelton set: Similar to equality, for a singleton set of the form {`E`} the translator analyzes `E` to decide whether the expression is primitive. Consequently, the translator produces `Tmp = [T_E]` if `E` is primitive and `T_E, Tmp = [V_E]` vice versa. Via unification the framework takes care to link the freshly introduced temporary variable to the identifier or expression it is meant to be assigned to.
Set enumeration: A set of arbitrary cardinality of the form {`E, F, ...`} follows the pattern of the singleton set, which is expressed by recursive application.
Ordered pair: The framework distinguishes between four cases for an ordered pair `E |-> F`.

1. If `E` and `F` are primitive: `Tmp = t(`T_E`, `T_F`)`.
2. If `E` is primitive and `F` is not: T_F`, Tmp = t(`T_E`, `V_F`)`.
3. If `E` is non-primitive and `F` is primitive: T_E`, Tmp = t(`V_E`, `T_F`)`.
4. If `E` and `F` are both non-primitive: T_E`, `T_F`, Tmp = t(`V_E`, `V_F`)`.

Set comprehension: Let $x_1,\ldots,x_i$ be all variables occurring in `P` and let `z =` $z_1,\ldots,z_j$ be the list of variables constrained by `P`. Let $y_1,\ldots,y_k$ with `i = j+k` denote external variables that occur in `P` but not in `z`. The set comprehension {`z|P`} is the set of every value of `z` that satisfies `P` and is translated to the goal `subconst1(Tmp,` $y_1,\ldots,y_k$`)`. Figure 2 shows the two introduced rules. `subconst1/k+1` creates a local scope, where the variables of `z` are unable to clash with the variables of the parent scope. s(CASP)'s built-in predicate `findall/3` is used to store in `Tmp` every instance of Ψ satisfying the condition T_P. Ψ is an ordered pair of the form `t(`z_1`, t(`z_2`, t(..., `z_j`)))` or scalar if `|z| = 1`.

```
subconst1(Tmp, y1,...,yk) :-
  findall(Ψ, subconst2(x1,...,xi), Tmp).
subconst2(x1,...,xi) :- TP.
```

Fig. 2: The two introduced rules for the set comprehension predicate.

Universal quantification: Let `z =` $z_1,\ldots,z_k$ be the list of variables constrained by `P`. Further, let $p_1,\ldots,p_i$ and $q_1,\ldots,q_j$ be the external variables that occur in `P` and `Q` respectively but do not occur in `z`. The predicate `!(z).(P => Q)` is translated to `subconst1(`$p_1,\ldots,p_i$`,` $q_1,\ldots,q_j$`)`. As this framework is restricted to finite domain declarations, it allows for using set comprehensions to express that `Q` is satisfied for each value of `z` satisfying `P`. Figure 3 depicts the three freshly introduced rules, where Ψ's definition is the same as for set comprehensions.

```
subconst1(p1,...,pi, q1,...,qj) :-
  findall(Ψ, subconst2(p1,...,pi, z1,...,zk), Tmp),
  for_all(Tmp, q1,...,qj).
subconst2(p1,...,pi, z1,...,zk) :- TP.
for_all([], q1,...,qj).
for_all([Ψ|Tmp], q1,...,qj) :-
  TQ,
  for_all(Tmp, q1,...,qj).
```

Fig. 3: The three introduced rules for the universal quantification predicate.

Arithmetical evaluation: This translation applies to the arithmetical operators $+, -, *, /, mod$. Without loss of generality, addition is selected to describe how the aforementioned operators are translated. s(CASP)'s CLP library is used in order to omit temporal restraints of conjunctions. The predicate `#=/2`, which subsumes and extends `is/2`, is used instead. The framework distinguishes between four cases for `m + n`.

1. If `m` and `n` are primitive: `Tmp #= T_m + T_n`.
2. If `m` is primitive and `n` is not: `T_n, Tmp #= T_m + V_n`.
3. If `m` is non-primitive and `n` is primitive: `T_m, Tmp #= V_m + T_n`.
4. If `m` and `n` are both non-primitive: `T_m, T_n, Tmp #= V_m + V_n`.

Union: `S \/ D` is translated to `T_S, T_D, union(V_S, V_D, Tmp)`. Except for the predicates genreal union and general intersection all other predicates are translated in the same way as union, i.e. intersection, difference, cartesian product, powerset, cardinality, (not) member, (not) (strict) subset, minimum, maximum, interval, relations, domain, range, composition, identity, domain/range restriction/subtraction, inverse, relational image, override, direct/parallel product, partial/total functions/injections/surjections, bijections and function application. For predicates of arity two, merely the translation of one of the arguments is omitted. The corresponding predicates are provided in an external file[1].
General union & general intersection: Essentially, the translator stacks successively as many `union/3` calls as necessary to compute the generalized union of the form `union(U)`. The code for a general intersection `inter(U)` is generated analogously. The framework distinguishes between three cases.

1. If `U = {S}`, then resolve `S` as a singleton set.
2. If `U = {S, D}`, then treat it as `S \/ D`.
3. If `n` $\geq$ `3`, then resolve $S_1, S_2 \in U$ as S_1 `\/` S_2 obtaining Tmp_1 and continue resolving Tmp_i and the next set S_{i+2} with `i` $<$ `n-1` as `T_{S_{i+2}}, union(Tmp_i, V_{S_{i+2}}, Tmp_{i+1})` until the final result Tmp_{n-1} is computed.

5 Empirical Evaluation

In this section I aim towards empirically evaluating the presented framework's capabilities by comparing benchmark performances of the new backend to the native ProB, Kodkod and Z3 backend. Regarding the translation's correctness test cases are embedded in ProB consisting of numerous exemplary expressions, which cover the supported predicates. For each test the s(CASP) backend first solves the underlying predicate obtaining a result, which is afterwards validated by ProB. Of course, this does not prove the backend's correctness.

In Figure 4 the runtime performances for the supported B predicates are presented. The result for a benchmark is obtained by solving and measuring the

[1] https://github.com/Alexandros31/B-to-sCASP/blob/main/preliminaries.pl

runtime of three separate synthetic B expressions on each backend (if available) via the ProB REPL. Additionally, the runtime of the plain s(CASP) engine is measured to gain more insight into s(CASP)'s performance by omitting the surrounding overhead introduced by the framework's processes. Each computation is executed three times with a one minute threshold on a freshly initialized REPL to counter inaccuracies in the measurements. The environment of this evaluation is a macOS 10.14.6 machine operating on a 7th generation i5 Intel processor at 3.1GHz. The displayed value representing a predicate's performance in milliseconds is obtained by averaging over the results of all three exemplary expressions, where every individual value is rounded half away from zero. The designation `n.a.` indicates that either a timeout or an "unsupported predicate exception" occurred. The benchmarks along with all performance measures for each run can be found on GitHub[2].

Predicate	**Native ProB**	**Kodkod**	**Z3**	**s(CASP)**	**Plain s(CASP)**
Comprehension	78	**39**	84	263	16
Univ. Quantifier	96	**40**	84	257	2
Union	67	**53**	71	310	47
Intersection	70	**52**	69	323	38
Difference	72	**54**	72	319	42
Cart. Product	76	**40**	137	293	5
Powerset	**74**	`n.a.`	126	1820	88
Cardinality	**50**	`n.a.`	59	241	1
Gen. Union	55	**42**	75	302	52
Gen. Intersection	56	**50**	71	311	61
Relations	**94**	`n.a.`	`n.a.`	698	39
Domain/Range	55	**31**	66	260	15
Composition	55	**30**	68	245	1
Identity	**67**	`n.a.`	81	249	0
Restrict/Subtract	54	**30**	67	256	14
Inverse	54	**41**	73	244	0
Relational Image	56	**42**	72	245	4
Override	**51**	`n.a.`	84	352	113
Direct Product	**54**	`n.a.`	70	249	6
Parallel Product	**55**	`n.a.`	106	280	12
Partial Functions	**76**	`n.a.`	`n.a.`	885*	333*
Total Functions	**75**	`n.a.`	`n.a.`	414*	161*
Partial Injections	**83**	`n.a.`	`n.a.`	537*	311*
Total Injections	**77**	`n.a.`	`n.a.`	1639*	1354*
Part. Surjections	**77**	`n.a.`	`n.a.`	379*	142*
Total Surjections	**78**	`n.a.`	`n.a.`	496*	256*
Bijections	**76**	`n.a.`	`n.a.`	2402*	2246*
Fun. Application	**54**	`n.a.`	77	241	0

Fig. 4: Comparison of predicate performances measured in milliseconds.

[2] https://github.com/Alexandros31/B-to-sCASP/blob/main/eval.md

The performance measurements of Figure 4 indicate that the native ProB, Kodkod and Z3 backend perform better compared to the s(CASP) backend, as their runtimes are generally faster. However, s(CASP) is able to obtain a result in some cases where Kodkod and Z3 are unable to follow. Furthermore, the plain s(CASP) engine's performances seem to be reasonable, as the corresponding runtimes are noticeably low in many cases. This suggests that the framework's overhead is considerably prevalent. On my machine I recorded an average startup time of 130 milliseconds for the Ciao engine. Compared to the best performing predicates the time of initialization renders roughly half of the entire process. However, these benchmarks are rather small compared to real-world examples. Hence, the time loss of the Ciao engine's initialization is more apparent.

In the following I express my thoughts on the gathered results. Note that these performances are heavily dependent on the efficiency of the custom predicates that are used for this translation, for instance `member/2` or `union/3`. The predicates set comprehension, universal quantifier, cartesian product, cardinality, domain/range, composition, identity, domain/range restriction/subtraction, inverse, relational image, direct product, parallel product and function application share the fastest runtimes among this evaluation as they succeed in under 300 milliseconds. Cartesian product, cardinality, composition, identity, inverse, relational image, direct product, parallel product and function application merely iterate over a list without any demanding additional task along the way, which presumably leads to consuming less time. The predicates domain/range and domain/range restriction/subtraction make use of `member/2` and set comprehension and universal quantification use s(CASP)'s built-in `findall/3` predicate. I assume that these specific benchmarks are not particularly challenging as further analysis suggests that the usage of `member/2` and `findall/3` leads to weaker performances.

Since s(CASP) lacks the cut operator, additional rules containing the goal `not member/2` are needed to correctly express some predicate's behavior such as `union/3` in this implementation. Consequently, in the instance of an element not being a member of the underlying list the rule invoking the negated goal is called nevertheless. This may induce further loss of time, especially for large lists. The other predicates, which mainly rely on `member/2` are union, intersection, difference, general union, general intersection and override. These predicates perform solidly relative to the aforementioned better performing ones.

The remaining predicates, i.e. powerset, relations, partial functions, total functions, partial injections, total injections, partial surjections, total surjections and bijections all invoke `findall/3`. These results show that the usage of `findall/3` poses a considerable bottleneck for this framework's performance. The asterisk indicates that the benchmark's largest expression is not succeeding for the given threshold of one minute. For these particular benchmarks only the remaining two expressions are used including ProB. Considering the lacking efficiency the framework offers a run option called "optimize", which lets ProB evaluate ground expressions before passing them to the framework.

Predicate	Native ProB	Kodkod	Z3	s(CASP)	Plain s(CASP)
4-Queens	115	**36**	282	369	31
5-Queens	112	**36**	285	438	37
6-Queens	117	**37**	1081	4927	3101
7-Queens	107	**39**	1262	30507	5190

Fig. 5: Comparison of n-queens performances measured in milliseconds.

Figure 5 shows the performances of various ProB backends dealing with different sizes of the n-queens problem. Since this implementation seems to struggle with the computation of functions, these measurements are supported by ProB with the "optimize" feature enabled.

Overall, the s(CASP) backend's performance is not on par with the other ones. In particular, the s(CASP) backend is outclassed by the native ProB, Kodkod and Z3 backend in every single benchmark. Nevertheless, in some cases s(CASP) succeeds while Kodkod and Z3 are unable to solve the given constraint. Yet, even in those instances the native ProB backend poses a superior option. However, one needs to consider that this framework hardly incorporates any optimizations, i.e. predicates are straightforwardly translated in most cases.

6 Future Work and Conclusion

Even though the presented framework is capable of translating a large portion of the B realm to s(CASP), there are some predicates left to be supported. Predicates such as iteration, closure, projection, lambda abstraction and the support of sequences should be included. Also enabling one to assign variables to infinite domains is imperative to be able to express more complex constraints. As s(CASP) offers a CLP backend and shares similarities with Prolog, this probably could be done similarly to how ProB handles those instances. Furthermore, it is desirable to extend this framework so that the order of predicates in a conjunction is irrelevant. This could be done by analyzing the underlying AST within the framework, and thus generate appropriate code that is solely reliant on already evaluated predicates. Moreover, this work's evaluation indicates that improving the framework as a whole to reduce its surrounding overhead may lead to better performances. Reimplementing parts of the custom predicates that utilize `findall/3` in a more efficient way should also lead to a more promising backend in general.

In conclusion, this work poses a foundation for translating B constraints to s(CASP). The empirical results indicate that the implemented backend is not quite on par with the other ones. However, I assume that this backend could be rendered more beneficial with further development. Furthermore, the presented backend can be used for verification of other employed solvers. Similar to how the s(CASP) backend is tested by ProB, ProB's and other solver's answers could thus be verified by s(CASP). Especially, for predicates that are solely supported by the native backend, the s(CASP) backend can be applied.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, New York, NY, USA (1996)
2. Arias, J., Carro, M., Salazar, E., Marple, K., Gupta, G.: Constraint answer set programming without grounding. Theory and Practice of Logic Programming **18**(3-4), 337–354 (2018)
3. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-Garcıa, P., Puebla, G.: The Ciao prolog system. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM) **95**, 96 (1997)
4. Carlsson, M., Widen, J., Andersson, J., Andersson, S., Boortz, K., Nilsson, H., Sjöland, T.: SICStus Prolog User's Manual, vol. 3. Swedish Institute of Computer Science, Kista, Sweden (1988)
5. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
6. Krings, S., Leuschel, M.: SMT solvers for validation of B and Event-B models. In: International Conference on Integrated Formal Methods. pp. 361–375. Springer (2016)
7. Leuschel, M., Butler, M.: ProB: A model checker for B. In: FME 2003: Formal Methods. vol. 2805, pp. 855–874. Springer, Berlin, Heidelberg (Sep 2003)
8. Leuschel, M., Butler, M.: ProB: An automated analysis toolset for the B method. International Journal on Software Tools for Technology Transfer **10**(2), 185–203 (Mar 2008)
9. Lifschitz, V.: Answer set programming. Springer Berlin (2019)
10. Marple, K., Salazar, E., Chen, Z., Gupta, G.: The s(ASP) predicate answer set programming system. The Association for Logic Programming Newsletter (2017)
11. Plagge, D., Leuschel, M.: Validating B, Z and TLA+ using ProB and Kodkod. In: International Symposium on Formal Methods. pp. 372–386. Springer (2012)
12. Schmidt, J., Leuschel, M.: Improving SMT solver integrations for the validation of B and Event-B models. In: International Conference on Formal Methods for Industrial Critical Systems. pp. 107–125. Springer (2021)
13. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 632–647. Springer (2007)