The BioASP Library: ASP Solutions for Systems Biology

Martin Gebser, Arne König, Torsten Schaub, Sven Thiele
University of Potsdam, Germany
Philippe Veber
Institut Cochin, France

***Abstract*—Today's molecular biology is confronted with enormous amounts of data, generated by new high-throughput technologies, along with an increasing number of biological models available over web repositories. This poses new challenges for bioinformatics to invent methods coping with incompleteness, heterogeneity, and mutual inconsistency of data and models. To this end, we built the library BioASP, providing a framework for analyzing biological data and models with Answer Set Programming (ASP). Due to the expressive modeling language, the inherent tolerance of incomplete knowledge, and efficient solving engines, ASP has proven to be an excellent tool for solving a variety of biological questions. The BioASP library implements methods for analyzing metabolic and gene regulatory networks, consistency checking, diagnosing, and repairing biological data and models. In particular, it allows for computing predictions and generating hypotheses about required expansions of biological models. To accomplish this, expert knowledge of both the biological application and the ASP paradigm needs to be combined. In fact, the functionalities provided by the BioASP library exploit technical know-how of modeling (biological) problems in ASP and gearing ASP solvers' parameters to them. Often, such best-practice technology is the result of an exhaustive series of tests. The BioASP library integrates our practical experience and offers them via easy-to-use Python functions, thus enabling ASP non-experts to solve biological questions with ASP.**

I. Introduction

The recent development of new high-throughput technologies in molecular biology has led to an enormous increase of measurable data. Furthermore, an increasing number of genetic and metabolic models for a variety of organisms are published in databases on the web, such as KEGG, Biomodels, Reactome, MetaCyc, and others. Despite of the huge amount of available information, biological data and models suffer from incompleteness and inconsistency. This makes it a non-trivial task to use available biological knowledge for drawing biologically meaningful conclusions in an automated way.

Due to its built-in tolerance of incompleteness, *Answer Set Programming* (ASP) [1], [2] has proven to be an excellent tool for solving a variety of questions on biological data and models. We utilize ASP for detecting and explaining inconsistencies [3], proposing repairs and computing predictions [4], as well as generating hypotheses about required expansions of biological models [5]. The obtained results can be used, e.g., for targeted literature research and experiment planning.

Solving biological questions often requires combining several computational steps and thus integrating ASP with traditional programming paradigms. For instance, we may have to parse and preprocess raw input data, then determine the optimum for an optimization problem, and eventually compute the intersection of all optimal solutions. Sometimes, we also need to combine different ASP solvers in a chain of computations. For example, we may first check the consistency of a biological model using the solver *clasp* [6] and then compute minimal diagnoses with the solver *claspD* [7]. Here, the different computational complexities of involved tasks (*NP* versus NP^{NP}) lead to distinct adequate solving approaches. Moreover, a solution to one subproblem may need to be fed back as input part of another subproblem. Last but not least, obtained results must be visualized in a user-friendly way.

To accommodate such complex solving scenarios, we created the library BioASP, written in Python. It provides functionalities for parsing biological inputs in several formats and transforming them into ASP facts, and it encapsulates the grounder *gringo* [8], [9] as well as the solvers *clasp* and *claspD*. In particular, encapsulating objects can be fed with logic programs describing different tasks, be launched with dedicated parameter settings, and pass on results for further processing. Thus, the BioASP library provides a framework to conveniently use ASP, embedded into the imperative programming paradigm of Python.

The outline of the paper is as follows. The next section provides a brief introduction to ASP. Section III outlines the architecture of the BioASP library. In Section IV, we describe the biological applications motivating the functionalities provided by BioASP and how such applications are solved using BioASP. Section V presents an available web service based on BioASP. In Section VI, we discuss related work and position our work. Finally, we conclude in Section VII.

II. Answer Set Programming

ASP is a declarative problem solving paradigm in which a problem is encoded by a collection of rules such that its intended models, called answer sets, represent solutions to the problem. ASP offers a rich yet easy modeling language [10], [8], [9] along with highly efficient inference engines based on state-of-the-art Boolean constraint solving technology [11], [6], [7]. We here only briefly summarize the essential of ASP; for detailed introductions, we refer the reader to [1], [2].

A *logic program* is a finite set of *rules* of the form

$$a_1; \ldots; a_l \leftarrow a_{l+1}, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n\ , \quad (1)$$

where a_i is an *atom* for $1 \leq i \leq n$. A rule r as in (1) is called a *fact* if $l=m=n=1$, and an *integrity constraint* if $l=0$. Let $head(r) = \{a_1, \ldots, a_l\}$, $body(r)^+ = \{a_{l+1}, \ldots, a_m\}$, and $body(r)^- = \{a_{m+1}, \ldots, a_n\}$ denote the *head* as well as the *positive* and *negative body* of r, respectively.

An interpretation is represented by the set of atoms that are true in it. A *model* of a logic program P is an interpretation in which all rules of P are true according to the standard definition of truth in propositional logic. Apart from letting ';' and ',' stand for disjunction and conjunction, respectively, this implies treating rules and default negation 'not' as implications and classical negation, respectively. Note that the (empty) head of an integrity constraint is false in every interpretation, while the empty body is true in every interpretation. Answer sets of P are particular models of P satisfying an additional stability criterion. Roughly, a set X of atoms is an answer set if, for every rule of form (1), X contains a minimum of atoms among $a_1, \ldots, a_l$ whenever $a_{l+1}, \ldots, a_m$ belong to X and no $a_{m+1}, \ldots, a_n$ belongs to X. However, the disjunction in heads of rules is, in general, not exclusive. Formally, an *answer set* X of a program P is a $\subseteq$-minimal model of

$$\{head(r) \leftarrow body(r)^+ \mid r \in P, body(r)^- \cap X = \emptyset\} \ .$$

As mentioned above, the idea of ASP is to encode a problem by a logic program such that its answer sets correspond to solutions to the problem. As common in logic programming, a program may contain first-order variables (over the implicitly given Herbrand universe). Thus, a rule containing variables is seen as the representative of all its variable-free instances. In practice, a program is first subject to instantiation, accomplished by a grounder like *gringo*. Depending on whether the obtained propositional program contains (proper) disjunctive rules, one then uses an appropriate solver to compute answer sets. We use for non-disjunctive programs the solver *clasp*, while *claspD* is needed to deal with the more complex class of disjunctive programs. All these systems are available at [12].

III. System Architecture

The BioASP library originates from our research in systems biology, where ASP has proven to be an effective tool for modeling and solving a variety of questions. However, to produce solutions based on ASP, it is often necessary to integrate it with existing environments and traditional programming paradigms. Python is a flexible and extensible scientific programming language used in various applications. For example, the BioPython project [13] provides solutions for transforming biological inputs into Python-utilizable data structures and offers interfaces to common bioinformatics programs. It implements tools to work with sequence data, performs standard machine learning tasks, and integrates with BioSQL, a sequence database schema also supported by the BioPerl [14] and BioJava [15] projects. In order to make the power of ASP accessible within an existing, rich system environment, BioASP provides classes encapsulating ASP tools: the grounder *gringo* as well as the solvers *clasp* and *claspD*. For the library to work, it is required that binaries of these systems are installed. In our biological applications, we

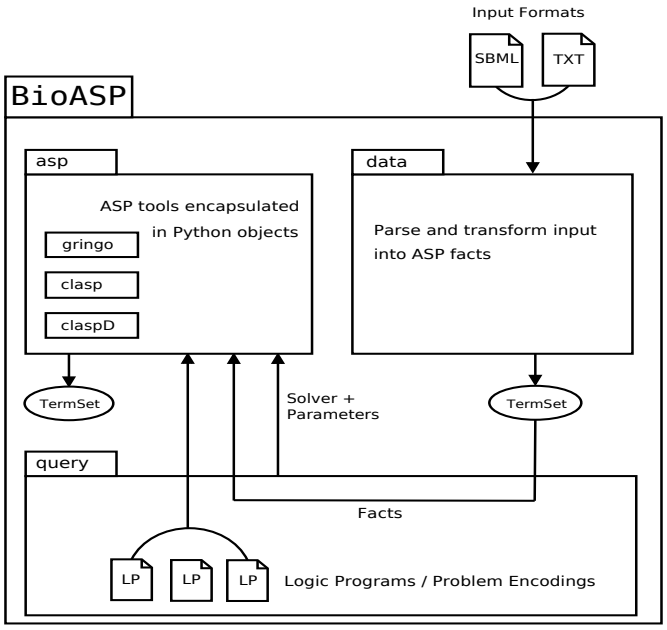

Fig. 1. Architecture of the BioASP library

are confronted with data in different formats, such as *SBML*, the Systems Biology Markup Language, and the *BioQuali* [16] format. The BioASP library provides functionalities to parse and transform these formats into ASP facts. For their implementation, BioASP utilizes the library *libSBML* [17] as well as the tool *ply*. As illustrated in Figure 1, the BioASP library consists of three main modules: the *data*, the *asp*, and the *query* module.

A. *The* data *module*

The *data* module implements functions to read and write different biological formats, such as *SBML* and the *BioQuali* format. These functions mainly parse input files and create a Python data structure *TermSet* containing the ASP facts representing a problem instance at hand. A *TermSet* can be joined with other *TermSet*s and finally be written to a file, suitable as input for the grounder *gringo*. The parsing functions are application-specific, as the facts they produce must match the atoms used in logic programs encoding questions on the input. If new formats or questions are to be addressed, new functions may need to be added for generating appropriate facts.

B. *The* asp *module*

The *asp* module constitutes the main component of the BioASP library. It provides the classes *GringoClasp*, *GringoClaspD*, and *GringoClaspOpt* that encapsulate the grounder *gringo* as well as the solvers *clasp* and *claspD*. These classes can be instantiated with dedicated parameter settings, and their objects can be run to solve logic programs in the input format of *gringo*. When such a solving process is finished, its result is returned as a *TermSet* or a list of *TermSet*s.

In more detail, an object of the class *GringoClasp* can be initialized with *gringo* options, such as constant definitions like `--const depth=10`, and *clasp* options, such as

`--heu=vsids` for setting the search heuristic. A solving process is then launched via the function `run(programs, nmodels)`, whose arguments are a list of input logic programs and an integer determining the maximum number of answer sets to be computed. The `run` function internally calls the grounder *gringo* to instantiate the input logic programs, pipes *gringo*'s output into *clasp*, parses the output of *clasp*, and finally returns answer sets as a list of *TermSet*s.

Due to their computational complexity, some of our biological problems are encoded by disjunctive logic programs. Since *clasp* cannot handle such programs, BioASP also provides the class *GringoClaspD*, which encapsulates *gringo* and the solver *claspD*. The class works similarly to *GringoClasp*, using *gringo* to instantiate logic programs but *claspD* instead of *clasp* for computing their answer sets.

The third class, *GringoClaspOpt*, is designed for dealing with optimization problems. Like *GringoClasp*, it encapsulates *gringo* and *clasp*, but it aims at logic programs containing optimization statements [8], [10]. Hence, *clasp* is here used to compute an optimal solution, and the associated optimum is returned by the `run` function.

The generic classes of the *asp* module can be utilized to solve a variety of problems, depending on the logic programs passed to the `run` functions of their objects.

C. *The* query *module*

The *query* module implements functionalities particular to biological questions. The provided functions work in a generic way on inputs given as *TermSet*s. The contained facts are combined with logic programs encoding the problem to be solved, using an appropriate object of a class provided by the *asp* module. Notably, the addressed biological question is taken into account for picking the parameter setting of such an object. Once they are computed, answer sets can be further processed, e.g., by filtering out atoms derived from facts.

IV. Application Areas

This section describes the biological questions that are currently addressed by the BioASP library. They stem from two major application areas in systems biology.

In our first application, we analyze gene regulatory networks and data on the variation of gene expressions from steady state shift experiments. Here, we rely upon the *Sign Consistency Model* [18], imposing constraints between experimental measurements and gene regulatory networks, modeled in terms of so-called influence graphs. Observed variations are represented as colorings on the nodes of an influence graph, as exemplified in Figure 2. For this setting, we developed logic programs to detect and explain inconsistencies between a network and experimental observations [3] and to compute minimal repairs allowing for prediction under inconsistency [4].

In our second application, we explore the biosynthetic capabilities of metabolic networks. Many metabolic networks are only partially defined, and only few metabolites can be identified without ambiguity. Our approach builds upon a formal method for analyzing large-scale metabolic networks [19], [20]. The basic idea is that a reaction can operate only if its reactants either are available as nutrients, referred to as seeds, or can be produced by other metabolic reactions. This allows for expanding a metabolic network by successively adding operable reactions along with their products. The set of metabolites in the resulting network represents all metabolites that can in principle be synthesized from seeds. For this setting, we developed logic programs to compute producible metabolites and minimal expansions of a network required to complete production pathways for target metabolites [5].

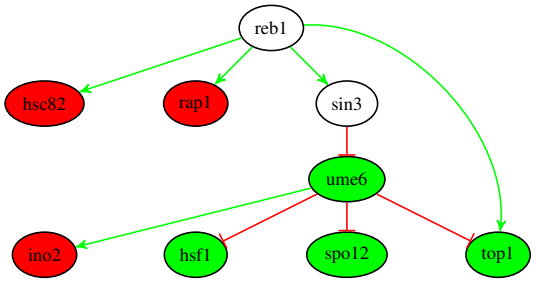

Fig. 2. Influence graph of genetic regulations in Escherichia Coli

In the following, we further detail the workflow of these two applications.

A. Diagnosing and Repairing on Gene Regulatory Networks

In the context of gene regulatory networks, we are interested in checking whether behaviors observed in experiments can be explained and in predicting behaviors of unobserved species. If experimental observations are inconsistent with a network, i.e., if they cannot be explained, minimal diagnoses can help to identify unreliable data or regulations missing in the network. Moreover, on the basis of minimal repairs, behaviors of unobserved species can be predicted even in the case of mutual inconsistency between network and data. Beyond analyzing available experimental data, the provided functionalities can be used for experiment planning. In this case, the input describes the desired behavior of a biological system, and predictions indicate conditions needed to achieve such behavior.

The BioQuali format has been designed for the textual representation of gene regulatory networks as well as data on the variation of gene expressions. For instance, Figure 3 shows the BioQuali representation of the partially colored influence graph in Figure 2. For such inputs, the BioASP library provides the functions `bioquali.readGraph(File)` and `bioquali.readProfile(File)` to parse a network and

```
reb1 -> hsc82 +
reb1 -> rap1 +
reb1 -> sin3 +
reb1 -> top1 +
sin3 -> ume6 -
ume6 -> ino2 +
ume6 -> hsf1 -
ume6 -> spo12 -
ume6 -> top1 -
```

```
hsc82 = -
rap1 = -
ume6 = +
ino2 = -
hsf1 = +
spo12 = +
top1 = +
```

Fig. 3. BioQuali format for networks (left) and observations (right)

```
obs_elabel(gen("reb1"),gen("hsc82"),1).
obs_elabel(gen("reb1"),gen("rap1"),1).
obs_elabel(gen("reb1"),gen("sin3"),1).
obs_elabel(gen("reb1"),gen("top1"),1).
obs_elabel(gen("sin3"),gen("ume6"),-1).
obs_elabel(gen("ume6"),gen("ino2"),1).
obs_elabel(gen("ume6"),gen("hsf1"),-1).
obs_elabel(gen("ume6"),gen("spo12"),-1).
obs_elabel(gen("ume6"),gen("top1"),-1).

exp("exp1").
obs_vlabel("exp1",gen("hsc82"),-1).
obs_vlabel("exp1",gen("rap1"),-1).
obs_vlabel("exp1",gen("ume6"),1).
obs_vlabel("exp1",gen("ino2"),-1).
obs_vlabel("exp1",gen("hsf1"),1).
obs_vlabel("exp1",gen("spo12"),1).
obs_vlabel("exp1",gen("top1"),1).
```

Fig. 4. ASP facts representing a network (top) and observations (bottom)

observations, respectively, and transform them into ASP facts. For example, the facts obtained from the statements in Figure 3 are shown in Figure 4. (The name `"exp1"` associated with experimental observations is obtained from the name of the file containing the observations in BioQuali format.)

For *consistency checking*, BioASP's *query* module provides the function `is_consistent(TermSet)`. It combines a problem instance like the one shown in Figure 4 with a logic program encoding the consistency check and uses a *GringoClasp* object for solving. If this object returns some answer set, network and observations are mutually consistent, and inconsistent otherwise.

In the case of consistency, we can further use the function `prediction_under_consistency(TermSet)` to *compute predictions*. It runs a *GringoClasp* object on the same input as used for consistency checking, but with the option `--cautious` of *clasp* being set. This makes *clasp* compute the intersection of all answer sets, corresponding to the predicted system behavior under the given observations. The atoms in the intersection that are not derived from facts (in the problem instance) are predicted and returned as a *TermSet*.

Otherwise, if network and observations are not consistent, one might be interested in the causes for inconsistency. To this end, BioASP's *query* module provides the function `compute_subset_mic(TermSet)`. It combines a problem instance like the one shown in Figure 4 with a logic program encoding diagnosis and uses a *GringoClaspD* object for solving. The obtained answer sets, representing *minimal*

```
[TermSet([active("exp1",gen("spo12"))]),
 TermSet([active("exp1",gen("hsf1"))]),
 TermSet([active("exp1",gen("hsc82")),
          active("exp1",gen("top1"))]),
 ...]
```

Fig. 5. Minimal diagnoses as a list of *TermSet*s

```
[TermSet([
  repair(vflip("exp1",gen("rap1"),-1)),
  repair(vflip("exp1",gen("ume6"),1)),
  repair(vflip("exp1",gen("hsc82"),-1))])
]
```

Fig. 6. A list containing a minimal repair as a *TermSet*

diagnoses of inconsistency, are returned as a list of *TermSet*s. For example, Figure 5 shows the first part of such a list.

Beyond diagnosing, BioASP supports several modes of repairing inconsistent networks and observations. A *repair* is understood as a collection of modifications on a network and observations that makes them mutually consistent. Such modifications can be the addition of regulations, flipping the polarity (activation or inhibition) or the observed variation (increase or decrease) of regulations or species, respectively, and allowing for species with unexplained variation. Since the adequacy of these modifications is application-specific, BioASP provides separate functions for different repair modes, confining the admissible kinds of modifications.

As the number of candidate repairs can be huge, one is usually not interested in all them, but only in minimal repairs, which modify network and observations as little as possible. Thus, the *query* module of BioASP provides functions to determine the minimum number of required modifications relative to repair modes. For instance, the function `card_minimal_repair_flip_obs(TermSet)` runs a *GringoClaspOpt* object to determine the minimum number of observations to be flipped for making a problem instance like the one shown in Figure 4 consistent. Then, the function `card_minimal_repair_flip_obs2(Opt,TermSet)` can be used to compute all minimal repairs by running a *GringoClasp* object with the options `--opt-val=Opt` and `--opt-all` of *clasp* being set. This second function returns a list of *TermSet*s containing the computed minimal repairs, such as the (singleton) list shown in Figure 6. The functions for other repair modes also follow the described scheme.

Complementing prediction in the consistent case, *prediction under inconsistency* can be accomplished by intersecting all answer sets comprising a minimal repair. To this end, the function `prediction_card_min_repair(Opt,TermSet)` runs a *GringoClasp* object on the same input as used for computing minimal repairs, but with the option `--cautious` of *clasp* being set in addition to `--opt-val=Opt` and `--opt-all`. As in the case of consistency, the atoms in the

```
TermSet([
 vlabel("exp1",gen("reb1"),1),
 vlabel("exp1",gen("hsc82"),1),
 vlabel("exp1",gen("rap1"),1),
 vlabel("exp1",gen("sin3"),1),
 vlabel("exp1",gen("ume6"),-1)
])
```

Fig. 7. Predicted variations as a *TermSet*

intersection of all answer sets (comprising a minimal repair) that are not derived from facts (in the problem instance) are predicted and returned as a *TermSet*, such as the one shown in Figure 7. Since the returned atoms hold under all minimal repairs relative to a repair mode, they describe system behavior that can sensibly be predicted even though the system is not globally consistent.

In summary, the functions provided by the BioASP library implement a variety of reasoning tasks on gene regulatory networks, modeled by influence graphs, and experimental data, described by observed variations. Given that some functionalities build on top of others, composite tasks can be accomplished by chaining several function calls in a workflow.

B. Metabolic Network Expansion

Our second application deals with the biosynthetic capabilities of metabolic networks. Given a set of nutrients, called seeds, and target metabolites that cannot be produced by reactions in a network, the goal is to generate hypotheses about expansions with additional reactions completing production pathways for the targets. Candidate reactions that can potentially be added are obtained from related networks available in web repositories, such as KEGG and MetaCyc.

The metabolic networks, seeds, and target metabolites used in our application are described in SBML format. To parse and transform such inputs into ASP facts, the BioASP library provides the functions `ReadSBMLnetwork(File,Name)`, `ReadSBMLseeds(File)`, and `ReadSBMLtargets(File)`. Exemplary facts obtained by them are shown in Figure 8.

For identifying unproducible targets, BioASP provides the function `get_unproducible(TermSet)`, which uses a *GringoClasp* object along with logic programs encoding this task. If there are unproducible targets, minimal expansions that complete production pathways for them are of interest. Thus, the function `get_extension_minimum(TermSet)` runs a *GringoClaspOpt* object to determine the minimum number of reactions to be added. Since there often are plenty minimal expansions in this application, it is useful to investigate their common fragments. To this end, the function

```
reaction("rea03981","ecoli").
reactant("com01126","rea03981").
product("com05702","rea03981").
reaction("rea09430","ecoli").
reactant("com05702","rea09430").
reactant("com00462","rea09430").
product("com03190","rea09430").
seed("com05702").
target("com03190").
target("com04283").

reaction("rea04982","metacyc").
reactant("com03190","rea04982").
reactant("com05702","rea04982").
product("com04283","rea04982").
```

Fig. 8. ASP facts representing metabolic networks, seeds, and targets

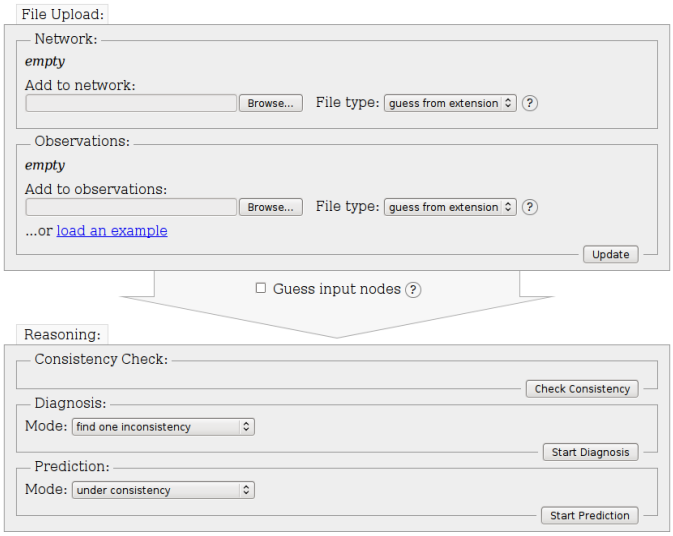

Fig. 9. Web interface for consistency checking, diagnosis, and prediction

`get_cautious_4_opt_extensions(TermSet,Opt)` can be used to compute the intersection of all minimal expansions by running a *GringoClasp* object with the options `--opt-val=Opt`, `--opt-all`, and `--cautious` of *clasp* being set, as in the case of prediction under inconsistency in the previous application.

V. Web Service

On the basis of the BioASP library, we built a web service[1] providing easy access to functionalities used in our first application, dealing with inconsistencies in gene regulatory networks. The web service does not require any locally installed software on the user side other than a web browser. As shown in Figure 9, it provides the possibility to upload files describing networks and experimental data in the BioQuali format. Furthermore, predefined examples allow users to instantly experience the functionalities of the web service. These include consistency checking, diagnosis, and prediction.

While consistency checking simply results in a positive or negative answer, we offer three diagnosis modes: "find one inconsistency," "find all inconsistencies," and "approximate all inconsistencies." The first mode provides a single minimal diagnosis, and the second one retrieves all of them. Obtained minimal diagnoses can be viewed either textually or graphically, as shown in Figure 10. If there are several minimal diagnoses, they can also be displayed in a combined way joining subnetworks with common species, thus highlighting regions of inconsistency. The third diagnosis mode works by iteratively removing minimal inconsistent subnetworks until the residual network is found to be consistent. This approach has been used in previous work [16] and is also offered by our web service for comparison.

Finally, our web service allows for prediction "under consistency" and "under cardinality minimal repair." The first mode is used if network and data are mutually consistent,

[1] `http://data.haiti.cs.uni-potsdam.de/wsgi/app`

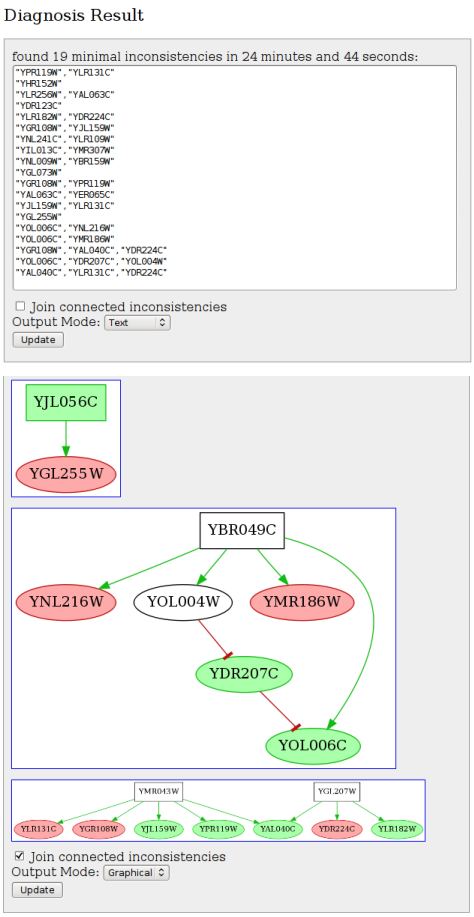

Fig. 10. Minimal diagnoses in textual (top) and graphical (bottom) mode

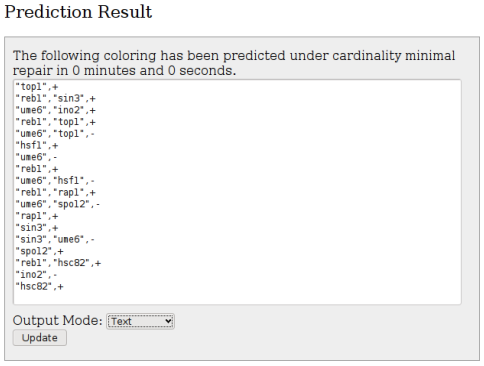

Fig. 11. Predictions in textual mode

and the second otherwise. In the latter case, the repair mode is currently fixed to modifying experimental observations by flipping increase to decrease, and vice versa. Hence, predictions are obtained only if the network as such (without any observations) is consistent. The computed predictions can be viewed textually, as shown in Figure 11.

VI. Related Work

Several libraries, such as BioPython [13], BioPerl [14], and BioJava [15], implement tools and methods for biological computations. To our knowledge, BioASP is the first library providing access to ASP-based methods in systems biology.

Biochemical networks can be modeled at different levels of abstraction. One usually distinguishes between quantitative (kinetic) and qualitative (stoichiometric or even purely causal) models. Quantitative models are applicable only if a certain amount of kinetic parameters is known, such as substance concentrations, equilibrium constants, or reaction rates. If enough information is available, a quantitative model (a set of differential equations) can be developed and analyzed with tools like CellDesigner [21] or GEPASI [22]. However, biological systems lacking detailed quantitative information are better analyzed with qualitative methods.

The BioASP library provides unique solutions for the qualitative analysis of metabolic and gene regulatory networks. Our approaches to diagnosis, repair, and prediction rely upon the Sign Consistency Model [18], which is also the basis of the BioQuali system [16]. BioQuali internally uses a solver for binary decision diagrams as inference engine. By virtue of exploiting logic programs and ASP solvers, the functionalities offered by BioASP (computing *all* minimal diagnoses, repairing, and predicting relative to repairs) go beyond the ones of BioQuali. Other qualitative methods for analyzing gene regulatory networks, e.g., [23], build on Thomas' approach [24]. Here, a system is represented by a state transition graph, and the goal is to analyze the dynamic behavior of the system, e.g., for identifying attractors and steady states.

Our approach to the analysis of metabolic networks is inspired by the network expansion algorithm due to Ebenhöh, Handorf, and Heinrich [19], [20]. They originally applied a stochastic method, determining and drawing conclusions from a representative yet incomplete set of samples of valid network expansions. Our approach additionally allows for the computation of minimal expansions and for reasoning over the complete set of *all* (minimal) expansions. Schuster et al. [25] use Petri nets for simulating metabolic networks and for computing elementary flux modes. A recent SAT-based approach by Soh and Inoue [26] aims at identifying necessary reactions in metabolic pathways.

VII. Conclusion

We presented the BioASP library as a framework providing ASP solutions for applications in systems biology. BioASP integrates with traditional programming paradigms to make the power of ASP accessible within an existing, rich system environment. Its current functionalities allow for analyzing metabolic and gene regulatory networks. The supported reasoning tasks include consistency checking, diagnosis, repair, prediction, and network expansion. The BioASP library is publicly available at `http://www.cs.uni-potsdam.de/bioasp/library/BioASP.tar.gz`. In the future, we want to address further biological applications and extend the BioASP library with functionalities for them.

Acknowledgments

This work was partially funded by the German Federal Ministry of Education and Research within the GoFORSYS project (`http://www.goforsys.org/`; grant 0313924).

REFERENCES

[1] C. Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[2] M. Gelfond, "Answer sets," in *Handbook of Knowledge Representation*, V. Lifschitz, F. van Hermelen, and B. Porter, Eds. Elsevier, 2008, pp. 285–316.

[3] M. Gebser, T. Schaub, S. Thiele, and P. Veber, "Detecting inconsistencies in large biological networks with answer set programming," *Theory and Practice of Logic Programming*, 2010, to appear.

[4] M. Gebser, C. Guziolowski, M. Ivanchev, T. Schaub, A. Siegel, S. Thiele, and P. Veber, "Repair and prediction (under inconsistency) in large biological networks with answer set programming," in *Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR'10)*, F. Lin and U. Sattler, Eds. AAAI Press, 2010, pp. 497–507.

[5] T. Schaub and S. Thiele, "Metabolic network expansion with ASP," in *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, P. Hill and D. Warren, Eds. Springer-Verlag, 2009, pp. 312–326.

[6] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "Conflict-driven answer set solving," in *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, M. Veloso, Ed. AAAI Press/MIT Press, 2007, pp. 386–392.

[7] C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. König, M. Ostrowski, and T. Schaub, "Conflict-driven disjunctive answer set solving," in *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, G. Brewka and J. Lang, Eds. AAAI Press, 2008, pp. 422–432.

[8] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele, "A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`," [12].

[9] M. Gebser, R. Kaminski, M. Ostrowski, T. Schaub, and S. Thiele, "On the input language of ASP grounder gringo," in *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, E. Erdem, F. Lin, and T. Schaub, Eds. Springer-Verlag, 2009, pp. 502–508.

[10] T. Syrjänen, "Lparse 1.0 user's manual," `http://www.tcs.hut.fi/Software/smodels/`.

[11] E. Giunchiglia, Y. Lierler, and M. Maratea, "Answer set programming based on propositional satisfiability," *Journal of Automated Reasoning*, vol. 36, no. 4, pp. 345–377, 2006.

[12] "Potassco, the Potsdam answer set solving collection, bundles tools for answer set programming developed at the University of Potsdam," `http://potassco.sourceforge.net/`.

[13] P. Cock, T. Antao, J. Chang, B. Chapman, C. Cox, A. Dalke, I. Friedberg, T. Hamelryck, F. Kauff, B. Wilczynski, and M. de Hoon, "Biopython: freely available Python tools for computational molecular biology and bioinformatics," *Bioinformatics*, vol. 25, no. 11, pp. 1422–1423, 2009.

[14] J. Stajich, D. Block, K. Boulez, S. Brenner, S. Chervitz, C. Dagdigian, G. Fuellen, J. Gilbert, I. Korf, H. Lapp, H. Lehväslaiho, C. Matsalla, C. Mungall, B. Osborne, M. Pocock, P. Schattner, M. Senger, L. Stein, E. Stupka, M. Wilkinson, and E. Birney, "The Bioperl toolkit: Perl modules for the life sciences," *Genome Research*, vol. 12, no. 10, pp. 1611–1618, 2002.

[15] R. Holland, T. Down, M. Pocock, A. Prlić, D. Huen, K. James, S. Foisy, A. Dräger, A. Yates, M. Heuer, and M. Schreiber, "BioJava: an open-source framework for bioinformatics," *Bioinformatics*, vol. 24, no. 18, pp. 2096–2097, 2008.

[16] C. Guziolowski, A. Bourde, F. Moreews, and A. Siegel, "BioQuali Cytoscape plugin: analysing the global consistency of regulatory networks," *BMC Genomics*, vol. 10, 2009.

[17] B. Bornstein, S. Keating, A. Jouraku, and M. Hucka, "LibSBML: an API library for SBML," *Bioinformatics*, vol. 24, no. 6, pp. 880–881, 2008.

[18] A. Siegel, O. Radulescu, M. Le Borgne, P. Veber, J. Ouy, and S. Lagarrigue, "Qualitative analysis of the relation between DNA microarray data and behavioral models of regulation networks," *Biosystems*, vol. 84, no. 2, pp. 153–174, 2006.

[19] O. Ebenhöh, T. Handorf, and R. Heinrich, "Structural analysis of expanding metabolic networks," *Genome Informatics*, vol. 15, no. 1, pp. 35–45, 2004.

[20] T. Handorf, O. Ebenhöh, and R. Heinrich, "Expanding metabolic networks: scopes of compounds, robustness, and evolution," *Journal of Molecular Evolution*, vol. 61, no. 4, pp. 498–512, 2005.

[21] A. Funahashi, Y. Matsuoka, A. Jouraku, M. Morohashi, N. Kikuchi, and H. Kitano, "CellDesigner 3.5: a versatile modeling tool for biochemical networks," *Proceedings of the IEEE*, vol. 96, no. 8, pp. 1254–1265, 2008.

[22] P. Mendes, "GEPASI: a software package for modelling the dynamics, steady states and control of biochemical and other systems," *Computer Applications in the Biosciences*, vol. 9, no. 5, pp. 563–571, 1993.

[23] H. de Jong, M. Page, C. Hernandez, and J. Geiselmann, "Qualitative simulation of genetic regulatory networks: method and application," in *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, B. Nebel, Ed. Morgan Kaufmann, 2001, pp. 67–73.

[24] R. Thomas and R. d'Ari, *Biological Feedback*. CRC Press, 1990.

[25] S. Schuster, T. Pfeiffer, F. Moldenhauer, I. Koch, and T. Dandekar, "Structural analysis of metabolic networks: elementary flux modes, analogy to Petri nets, and application to Mycoplasma pneumoniae," in *Proceedings of the German Conference on Bioinformatics (GCB'00)*, E. Bornberg-Bauer, U. Rost, and J. Stoye, Eds. Logos Verlag, 2000, pp. 115–120.

[26] T. Soh and K. Inoue, "Identifying necessary reactions in metabolic pathways by minimal model generation," in *Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI'10)*, H. Coelho, R. Studer, and M. Wooldridge, Eds. IOS Press, 2010, pp. 277–282.