Modular Representation of a Business Process Planner

Shahab Tasharrofi, Eugenia Ternovska

Simon Fraser University, Canada
{sta44,ter}@cs.sfu.ca

Abstract. The business process planner relies on external services for particular tasks. The tasks performed by each of the providers or the planner are often NP-complete, e.g. the Traveling Salesman Problem. Therefore, finding a combined solution is a computationally (as well as conceptually) complex task. Such a central planner could be used in business process management in e.g. logistics service provider, manufacturer supply chain management, mid-size businesses relying on external web services and cloud computing. The main challenge is a high level of uncertainty and that each module can be described in a different language. The language is determined by its suitability for the task and the expertise of the local developers. To allow for multiple languages, we approach the problem of finding combined solutions model-theoretically. We describe a knowledge representation formalism for representing such systems and then demonstrate how to use it for representing a business process planner. We prove correctness of our representation, describe general properties of modular systems and ideas for how to automate finding solutions.

1 Introduction

Formulating AI tasks as model finding has recently become very promising due to the overwhelming success of SAT (propositional satisfiability) solvers and related technology such as ASP (answer set programming) and SMT (satisfiability modulo theories). In our research direction we focus on a particular kind of model finding which we call *model expansion*. The task of model expansion underlies all search problems where for an instance of a problem, which we represent as a logical structure, one needs to find a certificate (solution) satisfying certain specification. For example, given a graph, we are looking for its 3-colouring in a classic NP-search problem. Such search problems occur broadly in applications; they include planning, scheduling, problems in formal verification (where we are looking for a path to a bug), computational biology, and so on. In addition to being quite common, the task of model expansion is generally simpler (for the same logic) than satisfiability from the computational point of view. Indeed, for a given logic $\mathcal{L}$, we have, in terms of computational complexity,

$$\text{MC}(\mathcal{L}) \leq \text{MX}(\mathcal{L}) \leq \text{Satisfiability}(\mathcal{L}),$$

where MC($\mathcal{L}$) stands for model checking (structure for the entire vocabulary of the formula in logic $\mathcal{L}$ is given), MX($\mathcal{L}$) stands for model expansion (structure interpreting a part of the vocabulary is given) and Satisfiability($\mathcal{L}$) stands for satisfiability task (where

we are looking for a structure satisfying the formula). A comparison of the complexity of the three tasks for several logics of practical interest is given in [15].

The next step is to extend the framework to a modular setting. In [21], we started to develop a model-theoretic framework to represent search problems which consist of several modules. In this paper, we develop our ideas further through an example of a Business Process Planner (BPP). This planner generalizes a wide range of practical problems. We envision such a planner used as a part of a multi-tool process management system. The task solved by BPP is extremely complex, and doing it manually requires significant resources. The technology is now ready to automate such computationally complex tasks, and our effort is geared towards making the technology available to less specialized users.

In systems like our planner, a high level of uncertainty is present. In our framework, we can model the following types of uncertainty.

- Each agent can see only the inputs and the outputs of other modules, but not their internals. The modules are viewed as black boxes by the outside world. Modules communicate with each other through common vocabulary symbols.
- Modules can be represented using languages that are not known to other modules. Such languages can even be old and no longer supported, as is common for legacy systems.
- Each module (an agent) can have multiple models (i.e., structures satisfying an axiomatization), each representing a possible plan of an individual module. This is a feature that generates uncertainty in planning. We view each module abstractly as a set of structures satisfying the axioms of the module.

The main challenge is that each module can be represented in a different language, reflecting the local problem's specifics and local expertise. Thus, the only way to formalize such a system is model-theoretic. Our goal is not only to formalize, but to eventually develop a method for finding solutions to complex modular systems like the BPP. This is a computationally complex task. Our inspiration for finding solutions to such systems comes from "combined" solvers for computationally complex tasks such as Satisfiability Modulo Theories (SMT). There, two kinds of propagation work interactively – propositional satisfiability (SAT) and theory propagation. In the case of modular systems, each module will have a so-called oracle that is similar to solvers/propagators used in SMT. If the logic language used by a module has a clear model-theoretic semantics, such an oracle (propagator) is easy to construct, but in the most extreme cases, derivations can be even performed by a human expert. At the level of solving, oracles would interact using a common internal solver language with a clear formal semantics. We believe that a formal model-theoretic approach is the right approach to developing a general algorithm for solving modular systems such as the BPP. This is another important motivation for developing a rigorous model-theoretic framework.

In this paper, we demonstrate how to use ideas of model expansion and modular systems together to naturally represent modular systems such as BPP. We prove correctness of our formalization and explain how finding solutions to such systems can be automated.

2 Business Process Planner

A business process planner is an entity which plans a particular task by relying on external services for particular tasks. Often, in business, there are cases when one needs to buy services from other service providers. The planner combines services provided by different companies to minimize the cost of the enterprise. The customer needs to allocate required services to different service providers and to ask them for their potential plans for their share. These plans will then be used to produce the final plan, which can be a computationally complex task. The tasks performed by each of the providers are often NP-complete, e.g. the Traveling Salesman Problem. Therefore, finding a combined solution is a computationally (as well as conceptually) complex task. Such a central planner could be used in business process management in many areas such as:

- **Logistics Service Provider** operates on the global scale, uses contracted carriers, local post, fleet management, driver dispatch, warehouse services, transportation management systems, e-business services as well as local logistics service providers with their own sub-modules.
- **Manufacturer Supply Chain Management** uses a supply chains planner relying on transportation, shipping services, various providers for inventory spaces, etc.. It uses services of third party logistics (3PL) providers, which themselves depend on services provided by smaller local companies.
- **Mid-size Businesses Relying on External Web Services and Cloud Computing** Such businesses often use data analysis services, storing, spreadsheet software (office suite), etc.. The new cloud-based software paradigm satisfies the same need in the domain of software systems.

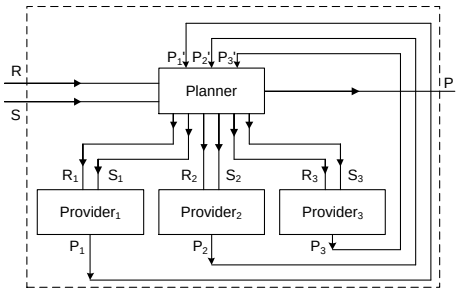

Fig. 1. Business Process Planner (BPP).

Figure 1 shows a general representation of a business process planner with three providers. Each of the solid boxes in Figure 1 represents a business entity which, while interested to participate in the process, is not necessarily willing to share the information that has affected their decisions. Therefore, any approach to representing and solving such systems that assumes unlimited access to complete axiomatizations of these entities is impractical.

The business process planner in Figure 1 takes a set S of services and a set R of restrictions (such as service dependencies or deadlines) and generates plan P. Each "Provider$_i$" takes a subset of services S_i and their restrictions R_i. Provider$_i$ generates a potential plan P_i for subset S_i of services and returns it to "Planner". Planner takes all these partial plans and, if not satisfied with them, reconsiders service allocations or providers. However, if satisfied, it outputs plan P by combining partial plans P_i.

3 Background: Model Expansion Task

In [17], the authors formalize combinatorial search problems as the task of *model expansion (MX)*, the logical task of expanding a given (mathematical) structure with new relations. Formally, the user axiomatizes their problem in some logic $\mathcal{L}$. This axiomatization relates an instance of the problem (a *finite structure*, i.e., a universe together with some relations and functions), and its solutions (certain *expansions* of that structure with new relations or functions). Logic $\mathcal{L}$ corresponds to a specification/modelling language. It could be an extension of first-order logic, or an ASP language, or a modelling language from the Constraint Programming (CP) community such as ESSENCE [12]. MX task underlies many practical approaches to declarative problem solving.

Recall that a vocabulary is a set of non-logical (predicate and function) symbols. An interpretation for a vocabulary is provided by a *structure*, which consists of a set, called the domain or universe and denoted by $dom(.)$, together with a collection of relations and (total) functions over the universe. A structure can be viewed as an *assignment* to the elements of the vocabulary. An expansion of a structure $\mathcal{A}$ is a structure $\mathcal{B}$ with the same universe, and which has all the relations and functions of $\mathcal{A}$, plus some additional relations or functions. The task of model expansion for an arbitrary logic $\mathcal{L}$ (abbreviated $\mathcal{L}$-MX), is:

Model Expansion for logic $\mathcal{L}$
Given: *(1)* An $\mathcal{L}$-formula ϕ with vocabulary $\sigma \cup \varepsilon$ and
(2) A structure $\mathcal{A}$ for σ
Find: an expansion of $\mathcal{A}$, to $\sigma \cup \varepsilon$, that satisfies ϕ.

We call σ, the vocabulary of $\mathcal{A}$, the *instance* vocabulary, and $\varepsilon := vocab(\phi) \setminus \sigma$ the *expansion* vocabulary[1].

Example 1. The following formula ϕ in the language of logic programming under answer set semantics constitutes an MX specification for Graph 3-colouring.

$$
\begin{aligned}
&1\{R(x), B(x), G(x)\}1 \leftarrow V(x).\\
&\bot \leftarrow E(x,y), R(x), R(y).\\
&\bot \leftarrow E(x,y), G(x), G(y).\\
&\bot \leftarrow E(x,y), B(x), B(y).
\end{aligned}
$$

An instance is a structure for vocabulary $\sigma = \{E\}$, i.e., a graph $\mathcal{A} = \mathcal{G} = (V; E)$. The task is to find an interpretation for the symbols of the expansion vocabulary $\varepsilon = \{R, B, G\}$ such that the expansion of $\mathcal{A}$ with these is a model of ϕ:

[1] By ":=" we mean "is by definition" or "denotes".

$$\underbrace{\overbrace{(V; E^{\mathcal{A}}}^{\mathcal{A}},\ R^{\mathcal{B}}, B^{\mathcal{B}}, G^{\mathcal{B}})}_{\mathcal{B}} \models \phi.$$

The interpretations of ε, for structures $\mathcal{B}$ that satisfy ϕ, are exactly the proper 3-colourings of $\mathcal{G}$.

Given a specification, we can talk about a set (class) of $\sigma \cup \varepsilon$-structures which satisfy the specification. Alternatively, we can simply talk about a set (class) of $\sigma \cup \varepsilon$-structures as an MX-task, without mentioning a particular specification the structures satisfy.

Example 2 (BPP as Model Expansion). In Figure 1, both the planner box and the provider boxes can be viewed as model expansion tasks. For example, the box labeled with "Provider$_1$" can be abstractly viewed as an MX task with instance vocabulary $\sigma = \{S_1, R_1\}$ and expansion vocabulary $\varepsilon = \{P_1\}$. The task is: given some services S_1 and some restrictions R_1, find a plan P_1 to deliver services in S_1 such that all restrictions in R_1 are satisfied.

Moreover, in Figure 1, the bigger box with dashed borders can also be viewed as an MX task with instance vocabulary $\sigma' = \{S, R\}$ and expansion vocabulary $\varepsilon' = \{P\}$. This task is a compound MX task whose result depends on the internal work of all the providers and the planner.

4 Modular Systems

This section presents the main concepts of modular systems.

Definition 1 (Primitive Module). *A* primitive module M *is a set (class) of $\sigma_M \cup \varepsilon_M$-structures, where σ_M is the instance vocabulary, ε_M is the expansion vocabulary.*

Each module can be axiomatized in a different logic. However, we can abstract away from the logics and study modular systems entirely model-theoretically.

A modular system is formally described as a set of primitive modules (individual sets of structures) combined using the operations of:

1. Projection($\pi_\tau(M)$) to restrict a module's vocabulary,
2. Composition($M_1 \rhd M_2$) to connect outputs of M_1 to M_2,
3. Intersection($M_1 \cap M_2$),
4. Union($M_1 \cup M_2$),
5. Feedback($M[R = S]$) which connects output S of M to its inputs R.

Formal definitions of these operations were introduces in [21] and are given below.

The initial development of of our algebraic approach was inspired by [14]. In contrast to that work, our contribution was to use a model-theoretic setting, simplify the framework and add a loop operator which increases the expressive power significantly, by one level in the polynomial time hierarchy. Here, we only consider modular systems that do not use the union operator.

Operations for Combining Modules

Definition 2 (Composable, Independent [14]). *Modules M_1 and M_2 are* composable *if $\varepsilon_{M_1} \cap \varepsilon_{M_2} = \emptyset$ (no output interference). Module M_1 is* independent *from M_2 if $\sigma_{M_1} \cap \varepsilon_{M_2} = \emptyset$ (no cyclic module dependencies).*

Definition 3 (Modular Systems). *Modular systems are built inductively from constraint modules using projection, composition, union and feedback operators:*

Base Case *A primitive module is a modular system.*

Projection *For modular system M and $\tau \subseteq \sigma_M \cup \varepsilon_M$, modular system $\pi_\tau(M)$ is defined such that (a) $\sigma_{\pi_\tau(M)} = \sigma_M \cap \tau$, (b) $\varepsilon_{\pi_\tau(M)} = \varepsilon_M \cap \tau$, and (c) $\mathcal{B} \in \pi_\tau(M)$ iff there is a structure $\mathcal{B}' \in M$ with $\mathcal{B}'|_\tau = \mathcal{B}$.*

Composition *For composable modular systems M and M' (no output interference) with M independent from M' (no cyclic module dependencies), $M \rhd M'$ is a modular system such that (a) $\sigma_{M \rhd M'} = \sigma_M \cup (\sigma_{M'} \setminus \varepsilon_M)$, (b) $\varepsilon_{M \rhd M'} = \varepsilon_M \cup \varepsilon_{M'}$, and (c) $\mathcal{B} \in (M \rhd M')$ iff $\mathcal{B}|_{vocab(M)} \in M$ and $\mathcal{B}|_{vocab(M')} \in M'$.*

Union *For modular systems M_1 and M_2 with $\sigma_{M_1} \cap \sigma_{M_2} = \sigma_{M_1} \cap \varepsilon_{M_2} = \varepsilon_{M_1} \cap \sigma_{M_2} = \emptyset$, the expression $M_1 \cup M_2$ defines a modular system such that (a) $\sigma_{M_1 \cup M_2} = \sigma_{M_1} \cup \sigma_{M_2}$, (b) $\varepsilon_{M_1 \cup M_2} = \varepsilon_{M_1} \cup \varepsilon_{M_2}$, and (c) $\mathcal{B} \in (M_1 \cup M_2)$ iff $\mathcal{B}|_{vocab(M_1)} \in M_1$ or $\mathcal{B}|_{vocab(M_2)} \in M_2$.*

Feedback *For modular system M and $R \in \sigma_M$ and $S \in \varepsilon_M$ being two symbols of similar type (i.e., either both function symbols or both predicate symbols) and of the same arities; expression $M[R = S]$ is a modular system such that (a) $\sigma_{M[R=S]} = \sigma_M \setminus \{R\}$, (b) $\varepsilon_{M[R=S]} = \varepsilon_M \cup \{R\}$, and (c) $\mathcal{B} \in M[R = S]$ iff $\mathcal{B} \in M$ and $R^{\mathcal{B}} = S^{\mathcal{B}}$.*

Further operators for combining modules can be defined as combinations of basic operators above. For instance, [14] introduced $M_1 \blacktriangleright M_2$ (composition with projection operator) as $\pi_{\sigma_{M_1} \cup \varepsilon_{M_2}}(M_1 \rhd M_2)$. Also, $M_1 \cap M_2$ is defined to be equivalent to $M_1 \rhd M_2$ (or $M_2 \rhd M_1$) when $\sigma_{M_1} \cap \varepsilon_{M_2} = \sigma_{M_2} \cap \varepsilon_{M_1} = \varepsilon_{M_1} \cap \varepsilon_{M_2} = \emptyset$.

Definition 4 (Models/Solutions of Modular Systems). *For a modular system M, a $(\sigma_M \cup \varepsilon_M)$-structure $\mathcal{B}$ is a* model of M *if $\mathcal{B} \in M$.*

Since each modular system is a set of structures, we call the structures in a modular system *models* of that system.

Example 3 (Stable Model Semantics). Let P be a normal logic program. We know S is a stable model for P iff $S = Dcl(P^S)$ where P^S is the reduct of P under set S of atoms (a positive program) and Dcl computes the deductive closure of a positive program, i.e., the smallest set of atoms satisfying it. Now, let $M_1(S, P, Q)$ be the module that given a set of atoms S and ASP program P computes the reduct Q of P under S. Also, let $M_2(Q, S')$ be a module that, given a positive logic program Q, returns the smallest set of atoms S' satisfying Q. Now define M as follows:

$$M := \pi_{\{P,S\}}((M_1 \rhd M_2)[S = S']).$$

Then, M represents a module which takes a ground ASP program P and returns all and only its stable models. Figure 2 shows the corresponding diagram of M.

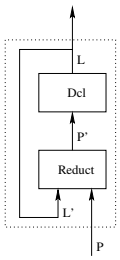

Fig. 2. Modular Representation of an ASP Solver.

On a model-theoretic level, this module represents all possible ASP programs and all their solutions, where programs are encoded by structures. While such a module is certainly possible, a more practical use would be where one module corresponds to a particular ASP program such as the one for graph 3-colouring in Example 1. Nevertheless, the Example 3 is useful because it represents a well-known construction and illustrates several concepts associated with modular systems.

Example 4 (BPP as a Modular System). Figure 1 can be viewed as a modular representation of the business process planner. There, each primitive module is represented by a box with solid borders and our module of interest is the compound module which is shown by the box with dotted borders. This module is specified by the following formula:

$$BPP := \pi_{\{S,R,P\}}(\text{Planner} \rhd ((\text{Provider}_1 \cap \text{Provider}_2 \cap \text{Provider}_3)[P'_1 = P_1][P'_2 = P_2][P'_3 = P_3])). \quad (1)$$

As in Figure 1, the only vocabulary symbols which are important outside the big box with dashed borders are S, R and P. There are also three feedbacks from P_1 to P'_1, P_2 to P'_2, and P_3 to P'_3.

5 Details of the Business Process Planner

In this section we give a detailed description of one of the many kinds of business process planners, i.e., a logistics service provider on the global scale which hires local carriers and warehouses. So, in Figure 1, "Planner" refers to the global entity and "Provider" refers to local entities.

The logistics provider need a plan to execute the services so that all restrictions are met. Some sample restrictions are: (1) latest delivery time (e.g., Halloween masks should be in stores before Halloween), (2) type of carrying vehicles (perishable products need refrigerator trucks), and (3) level of care needed (glass-works should be carried carefully).

We say that a plan P is good for a set of services S and restrictions R ($Good(P, S, R)$) if P does all services in S and satisfies all restrictions in R. For simplicity, here, we only consider time restrictions, i.e., the value of $t(i)$ is the (latest) delivery time for item i. There are also functions $s(.)$ and $d(.)$ to indicate the source

and the destination of an item. For an item i, a plan is a sequence of cities $\langle c_1, \cdots, c_n \rangle$ along with its pickup times $pt(i,j)$ and arrival time $at(i,j)$. So, we have that[2]:

$$\begin{aligned}
&\forall i \in Items\ (P(i) = \langle c_0, \cdots, c_n \rangle \supset \\
&\quad c_0 = s(i) \wedge c_n = d(i)), \\
&\forall i \in Items\ (P(i) = \langle c_0, \cdots, c_n \rangle \supset at(i,n) \leq t(i)), \\
&\forall i \in Items\ (P(i) = \langle c_0, \cdots, c_n \rangle \supset \\
&\quad \forall j \in [1,n]\ (connected(c_{j-1}, c_j)), \\
&\forall i \in Items\ (P(i) = \langle c_0, \cdots, c_n \rangle \supset \\
&\quad \forall j \in [0,n]\ (pt(i,j) \geq at(i,j))), \\
&\forall i \in Items\ (P(i) = \langle c_0, \cdots, c_n \rangle \supset \\
&\quad \forall j \in [1,n]\ (at(i,j) = pt(i,j-1) + time(c_{j-1}, c_j))).
\end{aligned}$$

Intuitively, these axioms tell us that a plan for each item should: (1) start at the source and end at the destination, (2) arrive at the destination sooner than their latest delivery time, (3) pass through cities which are connected to each other, (4) respect time constraints, i.e., be picked up at a city after they have arrived at that city, and (5) respect the distance between cities. Certainly, a good plan needs to satisfy all these conditions, but, of course, this does not give us a full axiomatization of the problem. Here, we do not even intend to do that, because we believe that this is enough for the reader to have a good idea on how such full axiomatizatins look like.

Given a definition of a good plan, one can define the intended solutions of a business process planner as below:

Definition 5 (Intended Solutions). *Let BPP be a business process planner with access to n providers. Structure $\mathcal{B}$ is an intended solution of BPP if:*

1. *$P^{\mathcal{B}}$ is good for $S^{\mathcal{B}}$ and $R^{\mathcal{B}}$, i.e., $\mathcal{B} \models Good(P,S,R)$,*
2. *All atomic actions A of $P^{\mathcal{B}}$ (here, moving items between different cities) are doable by one of the n providers.*

So, by Definition 5, if some set of services cannot be executed under some restrictions, there should not exist any solution for the whole modular system which interprets S by those services and R by those restrictions.

Now, to ensure that the intended solutions of modular system in Figure 1 coincide with the models of this modular system under our modular semantics, we use the declarative representations below for the modules:

[2] We slightly abuse logic notations here to keep the axiomatization simpler. For example, we use the notation $P(i) = \langle c_0, \cdots, c_n \rangle$ to denote that item i takes a path starting at city c_0 and then going to city c_1 and so on until it getting to city c_n. In practice, such a specification can be realized using two expansion function "$len(.)$" (to show the length of the path of an item) and "$loc(.,.)$" (to show its location). As an example, this is how the first axiom above is rewritten in terms of "len" and "loc":

$$\forall i \in Items\ (loc(i,0) = s(i) \wedge loc(i, len(i)) = d(i)).$$

Module "Planner" is the set of structures over vocabulary $\sigma = \{R, S, P_1, \cdots, P_n\}$ and $\varepsilon = \{P, S_1, \cdots, S_n, R_1, \cdots, R_n\}$ which satisfies:

$$Good(P, S, R) \Leftrightarrow \bigwedge_{i \in \{1, \cdots, n\}} Good(P_i, S_i, R_i), \tag{2}$$

$$P \text{ is a join of sub-plans } P_i (\text{for } i \in \{1, \cdots, n\}). \tag{3}$$

This module is easily specifiable in extended FO.

Module "Provider$_i$" is the set of structures over vocabulary $\sigma = \{R_i, S_i\}$ and $\varepsilon = \{P_i\}$ which satisfy $Good(P_i, S_i, R_i)$. Each such module "Provider$_i$" can be specified using mixed integer linear programming. Also, in practice, many such modules are realized using special purpose programs (so, no standard language). Our framework enables us to deal with such programs in a unified way.

Proposition 1 (Correctness). *Structure $\mathcal{B}$ is in modular system $BPP := \pi_{\{S,R,P\}}(\mathit{Planner} \rhd ((\mathit{Provider}_1 \cap \cdots \cap \mathit{Provider}_n)[P'_1 = P_1] \cdots [P'_n = P_n]))$ (where "Planner" and "Provider$_i$"s are defined as above) iff $\mathcal{B}$ is an intended solution of BPP (according to Definition 5).*

Proof. (1) Take $\mathcal{B}$ which satisfies all modules, each $P_i^{\mathcal{B}}$ has to be good for $S_i^{\mathcal{B}}$ and $R_i^{\mathcal{B}}$. Therefore, $P^{\mathcal{B}}$ is good for $S^{\mathcal{B}}$ and $R^{\mathcal{B}}$. Thus, $\mathcal{B}$ is an intended solution of BPP. (2) Conversely, take an intended solution $\mathcal{B}$. $P^{\mathcal{B}}$ should be such that $P^{\mathcal{B}}$ is good for $S^{\mathcal{B}}$ and $R^{\mathcal{B}}$. So, set $\mathcal{B}'$ to be an expansion of $\mathcal{B}$ such that $P_i^{\mathcal{B}'}$ is the parts of $P^{\mathcal{B}}$ which are executed by i-th provider. Also, $S_i^{\mathcal{B}'}$ is those services that $P_i^{\mathcal{B}'}$ executes and $R_i^{\mathcal{B}'}$ is those restrictions satisfied by $P_i^{\mathcal{B}'}$, e.g., the latest delivery time of item a is the delivery time of a according to $P_i^{\mathcal{B}'}$. Now, $P_i^{\mathcal{B}'}$ is good for $S_i^{\mathcal{B}'}$ and $R_i^{\mathcal{B}'}$. So, $\mathcal{B} \in BPP$.

6 The Bigger Picture

Complexity of the modular framework In this subsection, we summarize one of our important results about the modular framework from [21]. In order to do so, we first have to introduce the concepts of totality, determinacy, monotonicity, anti-monotonicity, etc. For lack of space, we do this through examples. The exact definitions can be found in [21].

Example 5 (Reachability). Consider the following model expansion task with $\sigma = \{S, E, B\}$ and $\varepsilon = \{R\}$:

$$\begin{array}{l} R(v) \leftarrow S(v). \\ R(v) \leftarrow R(u), E(u, v), \textbf{ not } B(u). \end{array} \tag{4}$$

where S represents a set of source vertices of a graph, E represents the edges of the graph, B represents a set of blocked vertices of the graph and R represents a set of vertices which can be reached from a source vertex without passing any blocked vertices.

Through this section, let M_R denote a primitive module which represents the MX task of Example 5. Obviously, $\sigma_{M_R} = \{S, E, B\}$ and $\varepsilon_{M_R} = \{R\}$: Then, we have:

Totality: Module M_R is $\{S, E, B\}$-$\{R\}$-total because for every interpretation of S, E and B, there is an interpretation for R which is a stable model of program 4.

Determinacy: Module M_R is $\{S, E, B\}$-$\{R\}$-deterministic because for every interpretation of S, E and B, there is at most one interpretation for R which satisfies (4).

Monotonicity: Module M_R is $\{E\}$-$\{S, B\}$-$\{R\}$-monotone because if we fix the interpretation of symbols S and B and increase the set of edges E, then the interpretation of R (reachable vertices) increases.

Anti-monotonicity: Module M_R is $\{E\}$-$\{S, B\}$-$\{R\}$-anti-monotone because if we fix the interpretation of S and E and increase the set of blocked vertices (B), then, the set R of reachable vertices decreases.

Polytime Checkability/Solvability: Module M_R is both polytime checkable (because one can check in polynomial time if a structure $\mathcal{B}$ belongs to M_R) and polytime solvable (because, given interpretations to S, E and B, one can compute the only valid interpretation for R in polynomial time). However, the module M_C which corresponds to the graph 3-coloring (Example 1) is polytime checkable but not polytime solvable (unless P=NP).

Now, we are ready to restate our main theorem from [21]. We should however point out one difference to the readers who are not accustomed to the logical approach to complexity: In theoretical computing science, a problem is a subset of $\{0,1\}^*$. However, in descriptive complexity, the equivalent definition of a problem being a set of structures is adopted. The following theorem gives a capturing result for complexity class NP:

Theorem 1 (Capturing NP over Finite Structures). *Let $\mathcal{K}$ be a problem over the class of finite structures closed under isomorphism. Then, the following are equivalent:*

1. *$\mathcal{K}$ is in NP,*
2. *$\mathcal{K}$ is the models of a modular system where all primitive modules M are σ_M-ε_M-deterministic, σ_M-total, σ_M-$vocab(\mathcal{K})$-ε_M-anti-monotone, and polytime solvable,*
3. *$\mathcal{K}$ is the models of a modular system with polytime checkable primitive modules.*

Note that Theorem 1 shows that when basic modules are restricted to polytime checkable modules, the modular system's expressive power is limited to NP. Without this restriction, the modular framework can represent Turing-complete problems. As an example, one can encode Turing machines as finite structures and have modules that accept a finite structure iff it corresponds to a halting Turing machine.

Theorem 1 shows that the feedback operator causes a jump in expressive power from P to NP (or, more generally, from Δ_k^P to Σ_{k+1}^P).

Example 6 (Stable Model Semantics). In Example 3, firstly, note that primitive module M_1 is $\{S\}$-total and $\{S\}$-$\{P\}$-$\{Q\}$-anti-monotone, and also polytime solvable. Secondly, module M_2 is $\{Q\}$-total, $\{Q\}$-$\{\}$-$\{S'\}$-monotone and, again, polytime solvable. However, the module $M := \pi_{\{P,S\}}((M_1 \rhd M_2)[S = S'])$ is neither total nor monotone or anti-monotone. Moreover, M represents the NP-complete problem of finding a stable model for a normal logic program. This shows how, in the modular framework, one can describe a complex modular system in terms of very simple primitive modules.

Solving modular systems We would like to find a method for solving complex tasks such as the application in this paper, without limiting to the particular structure of Figure 1, and without committing to a particular language. The language is determined by its suitability for the task and the expertise of the local developers. For example, the planner module is more easily specified as a SAT (propositional satisfiability) problem, while some provider modules are most easily specified using MILP (mixed integer linear programming), and global constraints with CP (constraint programming). A module performing scheduling with exceptions is more easily specified with ASP (answer set programming).

In our research, we focus on the central aspect of this challenging task, namely on *solving the underlying computationally complex task*, for arbitrary modular systems and arbitrary languages suitable for specifying combinatorially hard search/optimization problems. Our approach is model-theoretic. We aim at finding structures satisfying multi-language constraints of the modular system, where the system is viewed as a function of individual modules. *Our main goal is to develop and implement an algorithm that takes a modular system as its input and generates its solutions.* Such a prototype system should treat each primitive module as a black-box (i.e., should not assume access to a complete axiomatization of the module). Not assuming complete knowledge is essential in solving problems like business process planning.

We take our inspiration in how "combined" solvers are constructed in the general field of declarative problem solving. The field consists of many areas such as MILP, CP, ASP, SAT, and each of these areas has many solvers, including powerful "combined" solvers such as SMT, ASP-CP solvers. There are several methods e.g. cutting plain techniques of ILP, the formal interaction between SAT and theory solvers in SMT, etc. used in different communities. We made the fundamental observation [22] that while different on the surface, the techniques are similar when looked at model-theoretically. We proposed that those general principles can be used to develop a new method of solving modular systems as in the example above.

7 Related Work

In [21],we continued the line of research initiated in [14]. We introduced MX-based modular systems and extended the previous work in several ways such as adding the feedback (loop) operator, thus drastically increasing the expressive power. The current paper shows one of the important real-world applications of systems with loops. In our modelling of the business process planner, we use the language independence of modular systems in an essential way. This is an essential property because, in practice, providers use domain-specific software which may not belong to a well-studied logic. This property separates the modular framework of [21] from many other languages which support modularity such as modular logic programs [7, 18, 13], and frameworks with multiple languages [19, 10].

An early work on adding modularity to logic programs is [7]. There, the authors derive a semantics for modular logic programs by viewing a logic program as a generalized quantifier. This work is continued by [18] to introduce modular equivalence in normal logic programs under the stable model semantics. That work, in turn, is ex-

tended to define modularity for disjunctive programs in [13]. The last two papers focus on introducing modular programming in logic programs and dealing with difficulties that arise there.

Applications such as business process planning need an abstract notion of a module, independent from the languages used. Our MX-based modular framework is well-suited for this purpose. That cannot be said about many other approaches of adding modularity to ASP languages and FO(ID) (such as those described in [2, 1, 6]) because they address different goals.

Modular programming enables ASP languages to be extended by constraints or other external relations. This view is explored in [8, 9, 20, 3, 16]. While this view is advantageous in its own right, we needed an approach that is completely model-theoretic. Also, some practical modelling languages incorporate other modelling languages. For example, X-ASP [19] and ASP-PROLOG [10] extend prolog with ASP. Also ESRA [11], ESSENCE [12] and Zinc [5] are CP languages extended with features from other languages. Such practical modelling languages are further proof that combining different languages is extremely important for practitioners. We take this view to its extreme by looking at modules as only sets of structures and, thus, having no dependency on the language they are described in. The existing practical languages with support for specific languages could not have been applied to our task.

Yet another direction to modularity is the multi-context systems. In [4], the authors introduced non-monotonic bridge rules to the contextual reasoning and originated an interesting and active line of research followed by many others for solving or explaining inconsistencies in non-monotonic multi-context systems. However, we believe that this application cannot be naturally described as a multi-context system because it is impractical to define the concepts of a logic, a knowledge-base and an acceptability relation (these are concepts that are essential to define in multi-context systems) for a domain-specific application which might not use any known logical fragment.

8 Conclusion and Future Work

In this paper, we introduced an important range of real-world applications, i.e., business process planning. We discussed several examples of where this general scheme is used. Then we represented this problem as a model expansion task in the modular setting introduced in [21]. We gave a detailed description of the modules involved in describing business process planning in the modular framework and proved the correctness of our representation. Our main challenge is to devise an appropriate mathematical abstraction of "combined" solving. Remaining particular tasks include:

Algorithm Design and Implementation We will design and implement an algorithm that given a modular system, computes the models of that modular system iteratively, and then extracts the solutions.

Reduction in Search Space We will improve our algorithm by using approximation methods proposed in [21]. These methods correspond to least fixpoint and well-founded model computations (but in modular setting). We will extend our algorithm so that it prunes the search space by propagating information from the approximation process to the solver.

References

1. M. Balduccini. Modules and signature declarations for a-prolog: Progress report. In *Workshop on Software Engineering for Answer Set Programming (SEA 2007)*, pages 41–55, 2007.
2. Chitta Baral, Juraj Dzifcak, and Hiro Takahashi. Macros, macro calls and use of ensembles in modular answer set programming. In Sandro Etalle and Miroslaw Truszczynski, editors, *Logic Programming*, volume 4079 of *Lecture Notes in Computer Science*, pages 376–390. Springer Berlin / Heidelberg, 2006.
3. S. Baselice, P. Bonatti, and M. Gelfond. Towards an integration of answer set and constraint solving. In Maurizio Gabbrielli and Gopal Gupta, editors, *Logic Programming*, volume 3668 of *Lecture Notes in Computer Science*, pages 52–66. Springer Berlin / Heidelberg, 2005.
4. Gerhard Brewka and Thomas Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 1*, pages 385–390. AAAI Press, 2007.
5. Maria de la Banda, Kim Marriott, Reza Rafeh, and Mark Wallace. The modelling language zinc. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006*, volume 4204 of *Lecture Notes in Computer Science*, pages 700–705. Springer Berlin / Heidelberg, 2006.
6. M. Denecker and E. Ternovska. A logic of non-monotone inductive definitions. *Transactions on Computational Logic*, 9(2):1–51, 2008.
7. Thomas Eiter, Georg Gottlob, and Helmut Veith. Modular logic programming and generalized quantifiers. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Logic Programming And Nonmonotonic Reasoning*, volume 1265 of *Lecture Notes in Computer Science*, pages 289–308. Springer Berlin / Heidelberg, 1997.
8. Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *Proceedings of the 19th international joint conference on Artificial intelligence*, pages 90–96, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.
9. Islam Elkabani, Enrico Pontelli, and Tran Son. Smodels A – a system for computing answer sets of logic programs with aggregates. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 3662 of *Lecture Notes in Computer Science*, pages 427–431. Springer Berlin / Heidelberg, 2005.
10. O. Elkhatib, E. Pontelli, and T.C. Son. Asp – prolog: A system for reasoning about answer set programs in prolog. In *Proc. of Practical Aspects of Declarative Languages, 6th International Symposium, (PADL 2004)*, volume 3057, pages 148–162, Dallas, TX, USA, 2004.
11. Pierre Flener, Justin Pearson, and Magnus Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation*, volume 3018 of *Lecture Notes in Computer Science*, pages 214–232. Springer Berlin / Heidelberg, 2004.
12. Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13:268–306, September 2008.
13. Tomi Janhunen, Emilia Oikarinen, Hans Tompits, and Stefan Woltran. Modularity aspects of disjunctive stable models. *Journal of Artificial Intelligence Research*, 35:813–857, August 2009.
14. Matti Järvisalo, Emilia Oikarinen, Tomi Janhunen, and Ilkka Niemelä. A module-based framework for multi-language constraint modeling. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 5753 of *Lecture Notes in Computer Science*, pages 155–168. Springer Berlin / Heidelberg, 2009.

15. Antonina Kolokolova, Yongmei Liu, David Mitchell, and Eugenia Ternovska. On the complexity of model expansion. In *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 447–458, Berlin, Heidelberg, 2010. Springer-Verlag.
16. Veena Mellarkod, Michael Gelfond, and Yuanlin Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 53:251–287, 2008.
17. David G. Mitchell and Eugenia Ternovska. A framework for representing and solving np search problems. In *Proceedings of the 20th national conference on Artificial intelligence - Volume 1*, pages 430–435. AAAI Press, 2005.
18. Emilia Oikarinen and Tomi Janhunen. Modular equivalence for normal logic programs. In *Proceeding of the 2006 conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29 – September 1, 2006, Riva del Garda, Italy*, pages 412–416, Amsterdam, The Netherlands, The Netherlands, 2006. IOS Press.
19. T. Swift and D. S. Warren. *The XSB System*, 2009.
20. L. Tari, C. Baral, and S. Anwar. A language for modular answer set programming: Application to ACC tournament scheduling. In *Proc. of Answer Set Programming: Advances in Theory and Implementation*, CEUR-WS, pages 277–292, 2005.
21. S. Tasharrofi and E. Ternovska. A semantic account for modularity in multi-language modelling of search problems. In *FroCoS 2011*.
22. S. Tasharrofi, X. Wu, and E. Ternovska. Solving modular model expansion tasks. In *WLP/INAP 2011*.