EPiC Series in Computer Science

Volume 36, 2015, Pages 275–295

GCAI 2015. Global Conference on Artificial Intelligence

An Algebra of Combined Constraint Solving

Eugenia Ternovska

Simon Fraser University
ter@sfu.ca

Abstract

The paper describes a project aiming at developing formal foundations of combined multi-language constraint solving in the form of an algebra of modular systems. The basis for integration of different formalisms is the classic model theory. Each atomic module is a class of structures. It can be given, e.g., by a set of constraints in a constraint formalism that has an associated solver. Atomic modules are combined using a small number of algebraic operations. The algebra simultaneously resembles Codd's relational algebra, (but is defined on classes of structures instead of relational tables), and process algebras originated in the work of Milner and Hoare.

The goal of this paper is to establish and justify the main notions and research directions, make definitions precise. We explain several results, but do not give proofs. The proofs will appear in several papers. We keep this paper as a project description paper to discuss the overall project, to establish and bridge individual directions.

1 Introduction

Almost all non-trivial commercial software systems use libraries of reusable components. Component-based engineering is also widely used in VLSI circuit design, supported by a large number of libraries. The theory of combining conventional imperative programs and circuits is relatively well-developed. However, in knowledge-intensive computing, characterized by using so-called declarative programming, research on *programming from reusable components* is not very developed[1]. It would be very desirable to be able to take a program written in Answer Set Programming (ASP), combine it (say, as a non-deterministic choice) with a specification of a Constraint Satisfaction Problem (CSP), and then, sequentially, with an Integer Linear Program (ILP), and send feedback as an input to one of the first two programs. Such a programming method, from existing components, possibly found on the web, would be extremely useful. The main challenge however is that the programs may be written in different languages (even legacy languages), and rely on different solving technologies.

Recently, there has been a lot of work on technology integration. Examples include but are not limited to [1, 2, 3, 4, 5]. Combined solving is perhaps most developed in the SMT community, where theory propagations are tightly interleaved with satisfiability solving [6, 7]. Declarative and imperative types of programming sometimes have to be combined for best results.

Solving methods often rely on *decomposition* which are also studied in knowledge representation. A method for large knowledge base decompositions is developed by E. Amir and S. McIlraith [8].

[1] Related work that is most relevant is discussed at the end of the paper.

Tree decompositions are used to tame complexity in the work by S. Woltran and colleagues [9, 10]. Several similar notions of *tightening - relaxation*, *abstraction - refinement*, as well as several notions of *equivalence* are used in constraint solving and hierarchical program development.

For efficient solving, *special propagators* are identified and either implemented separately or integrated tightly into the main reasoning mechanism. For example, acyclicity [11] is added to SAT and ASP as a special propagator. BDDs are also used as special (and more powerful) propagators in hybrid constraint solving [12]. Experts can identify special sub-problems, which might still be combinatorially hard, but the solutions of which can help to solve the main problem. It is desirable to specify domain-specific propagators declaratively. A method for writing such specifications is proposed in our recent work [13].

Various kinds *of meta-reasoning* are used on top of knowledge-intensive methods. Meta-reasoning (high-level control) is used, for example, in dlvhex, a nonmonotonic logic programming system aiming at the integration of external computation sources and higher-order atoms [14]. Sophisticated control is used in [15], where the authors develop a method for lazy model expansion, where the grounding is generated lazily (on-the-fly) during search. The authors of [16] use bootstrapping for the IDP system, where inference engine itself is used to tackle some tasks solved inside a declarative system declaratively.

However, practical constraint solving remains a complex task. K. Francis noted, in [17]: "The results of the 2013 International Lightning Model and Solve competition served as a reminder that our tools are too hard to use. The winners solved the problems by hand, suggesting that creating an effective model was more time consuming (or daunting) than solving the problems."

Several research questions still need to be answered to bring solving and development technologies to the next level.

1. The technologies are diverse, some use declarative representations, some are purely imperative. What is common in all those advanced technologies, what would be a basis for integration? What are the units we combine, decompose into?
2. How can we combine those units? The language of combinations should be both expressive and feasible computationally.
3. What is the notion of a solution, how can such modular systems be solved?
4. What is (are) reasonable notion(s) of equivalence, modular system containment?

Our main goal is to provide a mathematical basis for combined constraint solving, and for specifying high-level control of such solving.

We believe that integration of technologies, from separate communities *cannot be done on the basis of one language*. Communities that develop technologies are attached to their languages. Experts usually get training (often during their PhD study) in one technology, and tend to keep applying it. It would be a utopia to assume that everyone would eventually switch to using a common representation language. It is also impractical to assume that axiomatization of each module would be translated to one language for solving. In such a translation, language-solver-specific efficiency adjustments in axiomatizations would be lost. Thus, a language-independent way of integration is needed. Since [18] we have been arguing that model theory is a good basis for integration in a language-independent way.

In this paper, we continue the line of research we started in [18] where Modular Systems were introduced. They were further developed in [19, 20, 21, 22, 23, 24, 25, 26]. *The main conceptual shift proposed in those works was to take model-theoretic approach seriously*. Due to the model-theoretic view, the formalism is language-independent, and combines modules specified in *arbitrary languages*.

A formalism for combining such programs should allow for arbitrary languages, and, at the same time, should lay foundations for solving combined systems, employing the solving power of solvers for languages used in the system. It should provide *heterogeneity in the syntax*, *homogeneity in the semantics*. Heterogeneity in the syntax is important because modules can be represented in different

languages. Homogeneity in the semantics is crucial for developing efficient solving algorithms. The first question is what a *module* is. In addition, a *small but expressive* set of operations for combining modules is needed. The formalism should have *controlled expressiveness.* On one hand, it should have *enough expressive power* to be able to produce interesting and useful combinations of modules, provide guarantees to be able to represent any problem, if its complexity is within the power of the language. On the other hand, the formalism should be *efficiently implementable*. Note that there is a tension between these two requirements. An increase in expressive power comes at the expense of efficiency. Finally, the formalism should contain a small number of operations, but sufficient for specifying high-level control.

In this paper, we define the notion of a module, which is language-independent, and describe an algebra for combining such modules, and consider its version with information propagation. We draw parallels with Codd's relational algebra, and process algebras originating from the work of Hoare and Milner. We discuss meta-solving, where the main solver is viewed as a "master" communicating with "slaves", individual modules that may have their own solvers associated with the language of the module (e.g., CSP, ASP modules). We argue, from the complexity-theoretic perspective, that such meta-solving is possible, for an interesting fragment, with the current technology for solving problems in the complexity class NP.

Several notions of inclusion (abstraction-refinement) and equivalence are possible for our algebra. We discuss two, one has more set-theoretic flavour, and the other is more behaviour-based.

The goal of the paper is to establish and justify the main notions, make definitions precise. We explain several results, but do not give proofs. We keep this paper as a project description paper to discuss the overall project, to establish and bridge individual directions. The paper contains two main parts. The first part, presented in Section 2, describes the general version of our algebra of modular systems. The second part, presented in Section 3, defines a directional variant of the algebra, where the information flow is specified.

2 Algebra of Modular Systems

A *vocabulary* (denoted, e.g. $\tau, \sigma, \varepsilon, \nu$) is a finite sequence of non-logical (predicate and function) symbols, each with an associated arity. A τ-structure, e.g. $\mathcal{A} = (A; S_1^{\mathcal{A}}, ..., S_n^{\mathcal{A}}, f_1^{\mathcal{A}}, ..., f_m^{\mathcal{A}}, c_1^{\mathcal{A}}, ..., c_l^{\mathcal{A}})$ is a domain A together with interpretations of predicate symbols, function and constants (0-ary functions) in τ. To simplify presentation, we view functions as a particular kind of relations and consider relational structures only. We use notations $vocab(\mathcal{A})$, $vocab(\phi)$, $vocab(M)$ to denote vocabulary of structure $\mathcal{A}$, formula ϕ and module M, respectively, and we use $\mathcal{B}|_\sigma$ to mean structure $\mathcal{B}$ restricted to vocabulary σ. Symbol $:=$ means "denotes" or "is by definition".

Informally, a **module** represents a piece of knowledge. Semantically, it corresponds to a class of structures. A module can be given by a knowledge base in some logic, be a specification of a robotic agent, be e.g. an ASP, CSP, ILP, CP program. It can even be a human making decisions. In practise, any decision procedure, of arbitrary complexity, could be used to specify a module.

Fix a vocabulary τ. An *atomic module symbol* (or simply an *atom*) is an expression $M(S_{i_1}, \ldots, S_{i_k})$, where $\{S_{i_1}, \ldots, S_{i_k}\}$ is called the *vocabulary* of M, denoted $vocab(M)$. For simplicity, we assume $vocab(M_i) \cap vocab(M_j) = \varnothing$ for $M_i \neq M_j$. We always have $vocab(M) \subseteq \tau$. We sometimes omit arguments from atomic module symbol expressions $M(S_{i_1}, \ldots, S_{i_k})$, and write simply M. We write $\overline{S}$ to denote tuple $\langle S_{i_1}, \ldots, S_{i_k} \rangle$.

The semantics of an atom $M(\overline{S})$ is given using an interpretation that is specific to that module. Such an interpretation indicates whether $M(\overline{S})$ is true (notation $\mathcal{B} \models_M M(\overline{S})$) or false on τ-structure $\mathcal{B}$ (notation $\mathcal{B} \not\models_M M(\overline{S})$). For any two τ-structures $\mathcal{B}_1, \mathcal{B}_2$ which coincide on $\{S_{i_1}, \ldots, S_{i_k}\}$, where $\{S_{i_1}, \ldots, S_{i_k}\} = vocab(M)$, we have $\mathcal{B}_1 \models_M M(\overline{S})$ iff $\mathcal{B}_2 \models_M M(\overline{S})$.

An *atomic module* (a semantic notion) that interprets atomic symbol $M(S_{i_1}, \ldots, S_{i_k})$ is a restriction of all structures where M is true to $vocab(M)$. In what follows, we will use atomic module symbols to refer to the corresponding atomic modules. That is, we will refer to modules by their names. For example, we will say "module M", where M is an atom.

Examples of Atomic Modules Represented in ASP and CSP

Modules can be given in many different ways. In particular, they can be axiomatized in classical first-order or higher-order logic.[2] It is harder to see how to represent modules in non-classical setting. We give examples of modules that encode well-known combinatorial problems, and show how those modules are representable in CSP and ASP.

The Constraint Satisfaction Problem (CSP) is: Instance: (V, D, C) where V is a finite set of variables, D is a set of *values* (also called domain), C is a finite set of *constraints* $\{C_1, \ldots, C_n\}$. Each constraint is a pair $(\overline{x}_i, R_i)$, where $\overline{x}_i$ is the *scope*, a list of variables of length m_i, and R_i is a m_i*-ary relation* over D. Question: Is there $f : V \to D$ such that $f(\overline{x}_i) \in R_i$ for all i? We call it the traditional AI form of CSP. One can also search for such f.

A homomorphism is a function $h : \mathcal{A} \to \mathcal{B}$ such that $\forall i[(a_1, \ldots, a_{n_i}) \in R_i^{\mathcal{A}} \Rightarrow (h(a_1), ..., h(a_{n_i})) \in R_i^{\mathcal{B}}]$. CSP in homomorphism form is: Instance: Two τ-structures, $\mathcal{A}$ and $\mathcal{B}$. Question: Is there a homomorphism $h : \mathcal{A} \to \mathcal{B}$?

We now explain a general translation from CSP instances in homomorphism form[3] to a representation as an atomic module (as a class of structures). An atomic module representing general CSP problem, M_{CSP}, where $\tau = \{R_1, \ldots R_l\}$, is a class of structures of the form

$$\mathcal{C} = (D; \underbrace{dom(\mathcal{A}), R_1^{\mathcal{A}}, \ldots R_l^{\mathcal{A}}}_{\mathcal{A}}, \underbrace{dom(\mathcal{B}), R_1^{\mathcal{B}}, \ldots R_l^{\mathcal{B}}}_{\mathcal{B}}, H^{\mathcal{C}}),$$

where D is an all-inclusive domain (it includes the domains of both structures), $dom(\mathcal{A})$, $dom(\mathcal{B})$ are new unary relations representing the domains of the two structures, and $H^{\mathcal{C}}$ is a relation representing homomorphism function h from $\mathcal{A}$ to $\mathcal{B}$.

We assume familiarity of the reader with Answer Set Programming. The main definitions and examples can be found, e.g. in [29]. We separate IDB and EDB predicates in each program, as is common in Datalog. Intuitively, EDB predicates are given by a database, and IDB predicates are definable in terms of those. For a program Π, we use Π' to denote an ASP program obtained by augmenting Π with ground atoms representing the database, the interpretations of the EDB predicates. We say a module is *representable in ASP* if there is an ASP program Π such that the stable models of Π', for each interpretation of the EDB predicates, when limited to the vocabulary of the module, are precisely the structures of the module.

In computer science, combinatorial decision problems are encoded as sets of binary strings. In finite model theory [30], such problems are represented as classes of structures. For example, 3-Colouring is represented by all graphs (which are structures) that are 3-colourable. We say a module M represents a problem if the structures for which the interpretation of M is true consist of the instance and the certificate of the problem, e.g., a graph and its colouring.

Example 1. Hamiltonian Circuit *In this example, τ is any vocabulary that includes $\{E, C_V\}$. An atomic module representing Hamiltonian Circuit problem denoted by $M_{\mathrm{HC}}(E, C_V)$, is a class of structures of the form $\mathcal{B} = (V; E^{\mathcal{B}}, C_V^{\mathcal{B}})$ such that $G = (V; E^{\mathcal{B}})$ is a graph, $C_V^{\mathcal{B}}$ is a cyclic permutation (i.e., an ordering that "returns" to its source) on V such that every pair of successive nodes in the ordering*

[2] Systems based on extensions of classical logic are IDP [27] and Enfragmo [28].

[3] To relate to a more traditional AI representation of CSP, think of elements in $\mathcal{A}$ as of variables. Tuples in relations in $\mathcal{A}$ correspond to constraint scopes, and elements in $\mathcal{B}$ to values. Relations in $\mathcal{B}$ are constraint relations.

are adjacent in G. The module is representable by a CSP instance $(\mathcal{C}, \mathcal{D})$ in homomorphism form where $\mathcal{C} = (V; E^{\mathcal{C}}, (\neq_V)^{\mathcal{C}})$, $\mathcal{D} = (V; E^{\mathcal{D}}, (\neq_V)^{\mathcal{D}})$ and there is a homomorphism H from $\mathcal{C}$ to $\mathcal{D}$. This CSP instance represents a class of structures of the form $\mathcal{E} = (V; E^{\mathcal{C}}, (\neq_V)^{\mathcal{C}}, E^{\mathcal{D}}, (\neq_V)^{\mathcal{D}}, H^{\mathcal{E}})$. Notice that $\neq_V$ and H are auxiliary, not a part of $M_{\mathrm{HC}}(E, C_V)$, and are specific to the CSP representation. Module $M_{\mathrm{HC}}(E, C_V)$ is also representable in ASP,[4] where relation $E^{\mathcal{B}}$ corresponds to symbol $edge$, relation $C_V^{\mathcal{B}}$ corresponds to $cycle$. Please see Remark 1.

$$\begin{array}{c} 1\,\{cycle(X,Y) : edge(X,Y)\}\,1 :- node(X). \\ 1\,\{cycle(X,Y) : edge(X,Y)\}\,1 :- node(Y). \\ reachable(Y) :- cycle(s,Y). \\ reachable(Y) :- cycle(X,Y), reachable(X). \\ :- node(X), not\ reachable(X). \end{array} \tag{1}$$

This ASP program is from page 89 of [29], where a detailed explanation can be found.

Remark 1. *Note that even though we start with a vocabulary for a module, and then define a module as a class of structures interpreting that vocabulary, a module is still a class of structures where each relation is just a relation, i.e., is "anonymous". There are two ways of connecting structures and vocabularies for those structures in model theory. The first way, used here, is to start with a vocabulary, and then define interpretations for that vocabulary. The second way is to start with structures $(D; R_1, \ldots R_n)$ and then introduce vocabulary for that structure by associating a vocabulary symbol with each relation. Our first submission of this paper followed the latter route, but was criticized by the reviewers as being hard to understand. Each way of dealing with this issue has its own advantages and disadvantages.*

Example 2. Betweenness *This problem was originally posed by Opatrny in 1979 [32]. Let τ be any vocabulary that contains $\{M, F\}$. An atomic module representing Betweenness problem, $M_{\mathrm{B}}(M, F)$, is a class of structures of the form $\mathcal{B} = (V; M^{\mathcal{B}}, F^{\mathcal{B}})$ where V is a finite set, $M^{\mathcal{B}} \subseteq V^3$, mapping $F^{\mathcal{B}} : V \to \mathbb{Q}$ is such that it generates a linear ordering of V such that, for each triple $(r, b, g) \in M^{\mathcal{B}}$, we have $r < b < g$ or $r > b > g$. It is representable by CSP instance $(V, \mathbb{Q}, C)$ in the traditional AI form, where V is a set variables, $\mathbb{Q}$ is the domain for those variables, and C is a set of constraints, $C = \{C_m \mid m \in M\}$, $C_m = ((u, v, w), R_b)$, $R_b = \{(a, b, c) \in \mathbb{Q}^3 \mid a < b < c \ \text{ or } \ a > b > c\}$.*

Example 3. k-Colouring *Vocabulary τ here contains al least $\{E, Col\}$. An atomic module representing k-Colouring problem, $M_{\mathrm{kCol}}(E, Col)$, is a class of structures of the form $\mathcal{B} = (V; E^{\mathcal{B}}, Col^{\mathcal{B}})$, where $Col^{\mathcal{B}}$ is a relation representing function that maps vertices to colours such that adjacent vertices are of different colour. In 3-Colouring module, $M_{3\mathrm{Col}}(E, Col)$, Col refers to a homomorphism $G^{\mathcal{B}} \to K_3$ expressed by binary relation $Col \subseteq V \times \{r, b, g\}$. Module $M_{3\mathrm{Col}}$ is also representable in ASP by the following program.*

$$\begin{array}{c} 1\{Col(X,r), Col(X,b), Col(X,g)\}1 : -V(X). \\ \bot : -Col(X,r), Col(Y,r), E(X,Y). \\ \bot : -Col(X,b), Col(Y,b), E(X,Y). \\ \bot : -Col(X,g), Col(Y,g), E(X,Y). \end{array} \tag{2}$$

Compound Modules

Compound modules are constructed from atomic ones using five basic operations and recursion. Our general strategy is to introduce a language as expressive as possible, while controlling the expressive power. This strategy guarantees that solvers can be constructed for interesting fragments of the resulting language, using atomic modules as oracles, as we proposed in [18].

[4] We give examples in ASP and CSP, but any other formalism, of arbitrary expressiveness, e.g. Essence [31], would work.

A **compound module** is defined by an algebraic expression $M = E(M_1, \ldots, M_s)$, where every atom has the form $M_i(S_{i_1}, \ldots, S_{i_k})$, and E is built by the grammar

$$E ::= \top \mid M_i | (E + E) | (E \times E) | (E - E) | \pi_\nu(E) | \sigma_\Theta(E) | Recursion. \tag{3}$$

Symbol $\top$ represents a tautological module that is true on every structure, and M_i are atomic module symbols. The operations (except recursion) are similar to Codd's relational algebra, but are of a higher order,[5] and are defined on *classes of structures* rather than on relational tables. Notice that the framework works for modules with infinite structures.

We intentionally leave recursion unspecified. $Recursion$ in (3) is a place holder. There is a discussion about recursion in a later section.

Remark 2. *We would like to emphasize that we are fully aware of stable model semantics [33] and the issues related to combining, e.g. $a :- not\ b.$, $b :- not\ c.$, $c :- not\ a.$. These issues can be resolved in modules internally, with the appropriate semantics. This is not what we are interested in in this version of the algebra. For us, each module is a black box, and its axiomatization is not visible from the outside. For modular systems under supported semantics and their connections to Multi-Context Systems [34], please see [23].*

Five Basic Operations (Semantics)

Satisfaction relation for compound algebraic expressions $\models_E$ is built inductively from module-specific satisfaction relations $\models_{M_i}$.

1. Product ($M_1(\overline{R}) \times M_2(\overline{S})$). For a structure $\mathcal{B}$ with vocabulary τ, we have $\mathcal{B} \models_{(M_1 \times M_2)} (M_1 \times M_2)(\overline{R}, \overline{S})$ iff both $\mathcal{B} \models_{M_1} M_1(\overline{R})$ and $\mathcal{B} \models_{M_2} M_2(\overline{S})$. That is, module $M_1 \times M_2$ represents a class of structures such that, when restricted to $vocab(M_i)$, it coincides with M_i, $i \in \{1, 2\}$.

2. Union ($M_1(\overline{R}) + M_2(\overline{S})$). For a structure $\mathcal{B}$ with vocabulary τ, $\mathcal{B} \models_{(M_1 + M_2)} (M_1 + M_2)(\overline{R}, \overline{S})$ iff at least one of $\mathcal{B} \models_{M_1} M_1(\overline{R})$ and $\mathcal{B} \models_{M_2} M_2(\overline{S})$ holds.[6]

3. Set Difference ($M_1(\overline{R}) - M_2(\overline{S})$). For a structure $\mathcal{B}$ with vocabulary τ, we have $\mathcal{B} \models_{(M_1 - M_2)} (M_1 - M_2)(\overline{R}, \overline{S})$ iff $\mathcal{B} \models_{M_1} M_1(\overline{R})$ and $\mathcal{B} \not\models_{M_2} M_2(\overline{S})$. Thus, $M_1 - M_2$ represents a class of structures over the joint vocabulary that are in M_1, but not in M_2.

4. Projection ($\pi_\nu(M(\overline{S}))$) is a family of unary operations, one for each ν. The semantics is given by $\mathcal{B} \models_{\pi_\nu(M)} \pi_\nu(M(\overline{S}))$ if $\mathcal{B} \models_M M(\overline{S})$. The operation restricts the structures of M to $\nu \subseteq vocab(M)$.

5. Selection is a family of unary operations of the form $\sigma_\Theta(M)$, where Θ is a condition that can be applied as a test to *each structure* of M[7]. It returns the subclass of M that satisfy condition Θ. The condition is an expression that is built up using $\wedge$, $\vee$, $\neg$, from equivalence operators $\equiv$, $\not\equiv$, applied to relation symbols in M and to interpretations of those (thus, we bring semantic elements into syntax in the latter case). E.g. $M_1.P \equiv M_2.S$ means that the interpretations of P (from M_1) and S (from M_2) are the same.[8] Selection is used to connect modules by equating relational symbols of equal arity.

The semantics is given by $\mathcal{B} \models_{\sigma_\Theta(M)} \sigma_\Theta(M)$ iff $\mathcal{B} \models_M M$ and $\mathcal{B} \models_{\mathrm{FO}} \Theta$, where $\models_{\mathrm{FO}}$ is the satisfaction relation in the standard first-order sense.

Remark 3. *Note that selection can be used to express* grounding*, the first stage of solving in e.g. ASP solvers. For that, Θ has to be of the form $(R_1 \equiv `R_1^{\mathcal{A}}') \wedge \cdots \wedge (R_n \equiv `R_n^{\mathcal{A}}')$, where we use single*

[5] For example, projection is onto relational rather than object variables.

[6] Note that in relational algebra, both arguments to the union and the difference must be relations of the same arity. Here, tuples $\overline{S}$, $\overline{R}$ can be of different length because interpretations are over τ-structures, where τ includes the vocabularies of all modules.

[7] The notation σ_Θ for selection clashes with the notation for instance vocabulary σ_M for module M, but the distinction is always clear because Θ is a special formula.

[8] Recall that we assumed, for simplicity, that $vocab(M_i) \cap vocab(M_j) = \varnothing$ for $M_i \neq M_j$.

quotation marks (‘, ’) to bring semantic elements, interpretations of predicates R_i in instance structure $\mathcal{A}$, into the syntax.

Remark 4. *One may argue that selection should not be an atomic operation but be represented by a module. This is certainly a possibility. The oracle for that new module would be a SAT solver. The situation here is similar to the situation with equality. One can either use equality as “built-in” or use the axioms of equality and run a theorem prover.*

Example 4. *The following algebraic expression specifies a combination of 3-Colouring and Hamiltonian Circuit.*

$$M_{3\mathrm{Col}-\mathrm{HC}}(E_1, Col) \ := \ \pi_{E_1,Col}(\sigma_{(M_{\mathrm{HC}}.C_V \equiv M_{3\mathrm{Col}}.E_2)}(M_{\mathrm{HC}}(E_1, C_V) \times M_{3\mathrm{Col}}(E_2, Col)). \quad (4)$$

Here, the selection operator requires the interpretations of C_V in M_{HC} and of E in $M_{3\mathrm{Col}}$ to be the same. Thus, Hamiltonian Circuit gets coloured. Projection hides the interpretation of C_V in M_{HC}, since it is the same as E_2's in $M_{3\mathrm{Col}}$, after selection is applied. Notice that the direction of information propagation is not specified, because it is not specified what the input is.

Example 5. *The following algebraic expression specifies a new module that is a class of graphs that are cycles.*

$$M_{\mathrm{Cycle}}(E) \ := \ \pi_E(\sigma_{(M_{\mathrm{HC}}.E \equiv M_{\mathrm{HC}}.C_V)}(M_{\mathrm{HC}}(E, C_V)).$$

Logic Counterpart of the Algebra

Just as relational algebra has a counterpart in relational calculus, our algebra has a counterpart in higher-order logic. The algebraic operations are expressible through the logic constructs of conjunction, disjunction, negation and second-order existential quantifier. We give an example to illustrate the idea. Expression (4) is represented in logic as

$$M_{3\mathrm{Col}-\mathrm{HC}}(E_1, Col) \ := \ \exists E_2 \ [(M_{\mathrm{HC}}(E_1, E_2) \wedge M_{3\mathrm{Col}}(E_2, Col)]. \quad (5)$$

Because of the model-theoretic approach, the formalism can be used as *multi-logic logic of modular systems*. For example, in the formula (5), one can simply substitute a specification of HC in brackets ($\{, \}$) for atomic module symbol $M_{\mathrm{HC}}(E_1, E_2)$ and a specification of 3Col for $M_{3\mathrm{Col}}(E_2, Col)$). Those specifications of atomic modules can be written in any formalism, e.g., CSP, ASP, ILP. The multi-logic logic was suggested in [18]. See also [26] and [22], where a concrete example is given.

Note that, to keep our presentation simple, we do not explain quantification (or projection) over object variables (those ranging over domain elements). Mathematically, such quantification is equivalent to second-order quantification over one-element sets, and can be mimicked by the second-order quantification we already have. In a more elaborate presentation of the formalism, quantification over object variables would be explained explicitly. Note that, similarly to relational variables, free object and function variables can also be used for communication between modules. Note also that while individual modules are already capable of solving optimization tasks (the optimum value can be given in one of the arguments), the least fixed point construct can generate the least value over a collection of modules combined in an algebraic expression.

From (4) and (5), the reader can observe the correspondence of the algebra to second-order logic. However, when we add information propagation to the algebra in the second part of the paper, we will also see that the algebraic view allows us to combine modules with information propagation into computational processes, like in process algebras. In addition, since we have recursion, the algebraic expressions are essentially Golog [35] programs.

Modular System Inclusion and Equivalence

Modular System Equivalence is one of the most important algorithmic tasks for modular systems. This is because in system development and rapid prototyping, it is useful to be able to replace a part of the system by an equivalent one, while preserving the overall functionality.

We say that two atomic or compound modules are *equal*,[9] denoted $M = M'$, if $\mathcal{B} \models_M M$ iff $\mathcal{B} \models_{M'} M'$. For example, $(M_1 + M_2) \times M_3 = (M_1 \times M_3) + (M_2 \times M_3)$, for any choice of atoms M_1, M_2 M_3. Another related notion is the notion of behavioural equivalence called *bisimulation*. We discuss this notion later in the paper, after we introduce a version of our algebra with information flow.

The notions of abstraction-refinement and tightening-relaxation are often used in reasoning about computational processes and operations research. We formalize a counterpart of those notions in the notion of *containment* for modular systems. We say that a M_1 is *contained* in M_2, denoted $M_1 \sqsubseteq M_2$, if $\mathcal{B} \models_{M_2} M_2$ is true whenever $\mathcal{B} \models_{M_1} M_1$ is true. In behavioural terms, we will talk about the notion of *simulation* (one direction of the bisimulation relation), when we discuss information flow.

This is essentially the problem of logical implication for our algebraic expressions. As we would expect, we have $M_1 = M_2$ iff $M_1 \sqsubseteq M_2$ and $M_2 \sqsubseteq M_1$. Modular system equivalence problem is undecidable in general. The proof is by reduction from Finite Validity problem. Thus, inclusion problem is also undecidable. However, it is decidable (NP-complete) for a broad subclass of modular systems.
A **conjunctive compound module (CCM)** is a module expressible by a relational algebra expression:

$$\pi_{\bar{R}}(\sigma_{\Theta}(M_1 \times \cdots \times M_n)),$$

where Θ is a conjunction of equivalence ($\equiv$) atomic formulas. CCMs are also expressible in logic form by formulas in prenex normal form built from atomic modules $M(R_1, \ldots, R_n)$, and $\wedge$ and $\exists$ (applied to relational symbols) only.

$$\exists S_1 \ldots \exists S_k \;\; \Phi(R_1, \ldots, R_n, S_1, \ldots, S_k),$$

where $\Phi(R_1, \ldots, R_n, S_1, , S_k)$ is a conjunction of atomic modules of the form $M(T_1, \ldots, T_l)$, and S_i, R_j, T_l are relational variables.

For conjunctive modules, Module Inclusion is NP-complete. We show it, using results about Homomorphisms, in a joint work with Andrei Bulatov. This is similar to the well-known theorem by A. Chandra and P. Merlin, from 1977 [36].

Recursion

One way to add recursion is to embed declarative languages axiomatizing modules (e.g, CSP, ASP, etc.) inside a general-purpose programming language, as K. Francis [17] proposed. Another way is to introduce recursion declaratively. This latter version is more similar to the declarative nature of our formalism. The declarative specification can later be refined, through hierarchical development, into a declarative or an imperative implementation.

In our case, recursion is either

1. a least fixed point construct, $\mu X_j E$, as in the Least Fixed Point logic FO(LFP) (see [37]), or
2. an extended Datalog program (in brackets($\{, \}$), *used within algebraic expressions*, similar to ID-logic [38].

[9] We use equality symbol (=) as a meta-symbol between algebraic expressions, and equivalence symbol ($\equiv$) as a logical connective (biconditional) in formulas.

The first option, $\mu X_j E$, will be used in the version of the algebra with information flow.[10] In the second option, the rule-based syntax of Datalog is more convenient for knowledge representation. The two versions are inter-translatable. Note that we do not overuse recursion. Recursion is limited to where it is needed computationally. The "classical" part allows for an easy component-wise replacement (strong equivalence holds).[11]

Monotonicity in fixed point computation is ensured by positive occurrences of atomic module symbols in the Datalog rules (or variables X_j in the $\mu X_j E$ construct).

Remark 5. *It is important to note that* we proceed cautiously with respect to adding expressive power, and this type of recursion adds at most polynomial time computation to the complexity of the modules. *For instance, logic programs under stable model semantics [33] and* dlvhex *programs [14] would be too expressive for our current goals. For related complexity results please refer to [39]. However, negation under well-founded semantics does not add more than a polynomial time computation, thus, such negation can be easily added. If complexity considerations are not important, one can definitely add negation as failure, and even use disjunctions in the heads of Datalog rules. Note that, if needed, our algebra can be used under supported semantics to mimic stable model semantics for classical logic [23]. However, that version is not used here.*

Our future work includes using the recursive construct to specify high-level control in solving combinatorially hard problems using dynamic programming and tree decomposition, along the lines of the work by S. Woltran and several collaborators [9, 10].

3 Version of the Algebra with Information Flow

Modules that have *inputs* and *outputs* are very common. For example, many software programs and hardware devices are of that form. In [40], we formalized the task of *model expansion*. In this task, a *given structure*, which might have an empty vocabulary, is expanded with interpretations of new vocabulary symbols to satisfy a specification. Complexity-wise, model expansion lies in-between model checking (full structure is given) and satisfiability (no structure is given). However, unlike those two tasks, model expansion has not been studied extensively. At the same time, it is very common in constraint solving. Current research trends show that studying it in the context of hybrid constraint solving is very important.

In this section, we propose a formalism to study this task in the context of modular systems. This formalism can be viewed as a *modal counterpart* of the algebra above, in the same way as modal logic is a fragment of first-order logic, and the modal mu-calculus μL is a fragment of FO(LFP) (see e.g. [41] for references).

There are significant benefits in studying modal fragments of our algebra. Such fragments possess good model-theoretic and algorithmic properties, and will lead to (1) automatic hybrid solver generation and (2) automatic generation of solvers with special-purpose propagators.

Model Expansion Task for Atomic Modules

For atomic modules that are specified in a logic-based formalism, one can talk about model expansion.
Given: a formula ϕ in logic $\mathcal{L}$ over vocabulary $\sigma \cup \varepsilon$, such that $\sigma \cap \varepsilon = \emptyset$ and σ-structure $\mathcal{A}$.
Find: structure $\mathcal{B}$ such that $\mathcal{B}|_\sigma = \mathcal{A}$ and $\mathcal{B} \models \phi$.

[10] Both version of the recursive construct can be used in the dynamic variant of the algebra (Section 3), however it is shorter to explain $\mu X_j E$.

[11] People who are more comfortable with a rule-based syntax, can ignore the "classical" part of the algebra and use only the Datalog construct, since it is a well-formed expression on its own.

We call σ *instance (input)* and ε *expansion (output)* vocabularies. Logic $\mathcal{L}$ corresponds to a specification/modelling language, e.g. ASP, CP, First-order logic with Inductive definitions [27]. The case $\sigma = \emptyset$ is called *model generation*. The case $\varepsilon = \emptyset$ corresponds to *model checking*, when full structure is given, and the task is to check if the formula is true in the structure. Note that at least the domain needs to be given, otherwise, say, for first-order logic, the task becomes that of satisfiability, which is undecidable.

Model expansion tasks are common in AI planning, scheduling, logistics, supply chain management, etc. Java programs, if they are of input-output type, can be viewed as model expansion tasks, regardless of what they do internally. ASP systems, e.g., Clasp [42] mostly solve model expansion, and so do CP languages such as Essence [31], as shown in [43]. Problems solved in ASP competitions are mostly in model expansion form. CSP in the traditional AI form (respectively, in the homomorphism form) is representable by model expansion where mappings to domain elements (respectively, homomorphism functions) are expansion functions.

As we mentioned above, selection operation can formalize grounding, thus connecting designated input symbols to an input structure. As a result, only structures that expand the instance structure are selected. This way selection specifies the *direction of information propagation*, from problem instances to expansions of the inputs. In fact *any* direction of information propagation can be specified if needed, e.g. from colours to graphs in 3-Colouring.

Example 6. Hamiltonian Circuit problem[12] *Given: a graph $G = (V; E^{\mathcal{G}})$. Find: a cyclic ordering of V such that every pair of successive nodes in the ordering are adjacent in G. This model expansion task is representable in the algebra by restricting the atomic module $M_{\mathrm{HC}}(E, C_V)$ to the given input graph, $G = (V; E^{\mathcal{G}})$. We write algebraic expression to represent this restriction as follows: $\sigma_{(E \equiv `E^{\mathcal{G}}')} M_{\mathrm{HC}}(E, C_V)$. This expression is equivalent to fixing interpretation of relational variable E, and leaving relational variable C_V free to interpretation. As a result, interpretations of the free symbols give us all possible Hamiltonian Circuits for the given graph.*

Example 7. k-Colouring problem *Given: a graph $G = (V; E^{\mathcal{G}})$ and a number k. Find: A colouring of V in k colours such that adjacent vertices are of different colour. The problem is tractable for $k = 2$, NP-complete for $k \geq 3$. This model expansion task is representable in the algebra by restricting the atomic module $M_{3\mathrm{Col}}(E, Col)$ to the given input graph, $G = (V; E^{\mathcal{G}})$. We write algebraic expression to represent this restriction as follows: $\sigma_{(E \equiv `E^{\mathcal{G}}')} M_{3\mathrm{Col}}(E, Col)$. Given graph G on the input, the interpretations of Col that satisfy the ASP program above (2) are exactly the proper 3-colourings of G.*

Note that the same axiomatization can be used in different ways. Say, in 3-Colouring, a colouring could be given, and one could be searching for possible edges.

As we saw in the examples, an atomic module M can be given through representing it by a formula ϕ in some logic (a formalism) $\mathcal{L}$ such as ASP,[13] CSP, such that $vocab(M) = vocab(\phi) = \sigma \cup \varepsilon_a \cup \varepsilon$. That is, ϕ may contain auxiliary expansion symbols ε_a that are different from the output symbols ε of M. (It may not even be possible to axiomatize M in that particular logic $\mathcal{L}$ without using any auxiliary symbols).

Modules as Non-Deterministic Operators

Model-theoretic view in our framework has a dual, operational view. The duality allows us to interpret atomic modules both *declaratively* (as models of axiomatizations) and *imperatively* (as elementary actions). Notice that, when input vocabulary is specified, each module (atomic or combined) can be

[12] While data complexity in these examples is NP-complete, it can be arbitrary in general.

[13] To use ASP representation of modules where the direction of information propagation is arbitrary (including where the interpretations of the heads of the rules are given on the input), stable model semantics has to be modified slightly, as noted in [44] where the authors introduce input answer sets.

viewed as a *non-deterministic operator*. The operator takes a structure that provides interpretation to the input vocabulary, and outputs expansions of that structure obtained by the internal laws of the module. Alternatively, we can view each module as a higher-order binary input-output relation.

From now on, we assume that input (σ_M), output (ε_M) vocabularies are assigned to some atoms M. We require $\sigma_M \cap \varepsilon_M = \emptyset$, $\sigma_M \cup \varepsilon_M \subseteq vocab(M)$. In algebraic expressions and corresponding logic formulas, we underline designated input symbols, i.e., those in σ_M. Output symbols are free (are not quantified). For example:

$$\exists E_2 \ [(M_{\mathrm{HC}}(\underline{E_1}, E_2) \wedge M_{3\mathrm{Col}}(E_2, Col)]. \tag{6}$$

The quantified symbol E_2 is not visible from the outside. The expansion (output) vocabulary of this compound modular system is $\varepsilon = \{Col\}$, the instance vocabulary is $\sigma_M = \{E_1\}$.

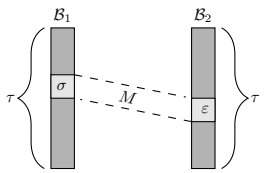

Figure 1: Modules as non-deterministic operators. Each $\sigma \cup \varepsilon$-structure in module M has interpretation of its σ part on the left, and of ε part on the right.

With this view on modules as non-deterministic operators, modular systems are *transition systems*, where states are τ-structures and transitions are "applications of modules". Figure 1 represents one of the possible non-deterministic transitions performed by module M. Module M "looks" at the interpretations of the input symbols σ_M in the structure $\mathcal{B}_1$ on the left, and expands $\mathcal{B}_1|_{\sigma_M}$ to produce interpretation of ε_M recorded in the structure $\mathcal{B}_2$ (one of the "M-successor" structures) on the right. Interpretations of all other symbols, including those in σ_M, stay the same, and get transferred from $\mathcal{B}_1$ to $\mathcal{B}_2$ by inertia. This is similar to the frame axioms in the situation calculus as described in R. Reiter's book [45]. Notice that, similarly to the situation calculus, inertia is applied to *atomic* modules only. Each structure (denoted by the dashed lines in the figure) in module M is composed from its σ_M part recorded in the structure $\mathcal{B}_1$ on the left, and its ε_M part recorded in the structure $\mathcal{B}_2$ on the right. Notice also that because of inertia, successor structure $\mathcal{B}_2$ contains information about both σ_M and ε_M part of a structure in M. Thus, the union of all M-successor structures, when limited to $vocab(M)$, is the entire module M.

To illustrate transitions using our examples, in (6), first $M_{\mathrm{HC}}(\underline{E_1}, C_V)$ makes transition by producing possibly several Hamiltonian Circuits. The interpretation of the expansion vocabulary, $\{C_V\}$ changes, everything else is transferred by inertia. Then each resulting structure is taken as in input to 3-Colouring, $M_{3\mathrm{Col}}(E_2, Col)$, where $\{C_V\}$ is "fed" to E_2, although this is hidden from the outside observer by the existential quantifier in (6). The second module produces non-deterministic transitions, one for each generated colouring.

A Dynamic Version of the Algebra

Here we take advantage of the view on modules as binary higher-order input-output relations. We focus on information-flow counterpart of the algebra, a version that may be called dynamic. We may also call it "a logic of hybrid MX tasks".

Fix a relational vocabulary τ and a vocabulary of atomic module symbols τ_{MS}. We require $vocab(M_i) \subseteq \tau$ for each $M_i \in \tau_{MS}$.

Let τ_P, where $\tau_P \subseteq \tau_{MS}$, be a vocabulary of atomic module symbols $M_i(S_{i_1}, \ldots, S_{i_k})$ where inputs are *not* specified. We call them *propositions*. Let τ_{act}, where $\tau_{act} \subseteq \tau_{MS}$, be a vocabulary of atomic module symbols $M_i(\underline{S_{i_1}}, \ldots, S_{i_k})$, where inputs are underlined. We call them *actions*. For one module

symbol M_i, we can potentially have both a proposition and several actions, depending on the choice of the inputs. For simplicity, we assume $vocab(M_i) \cap vocab(M_j) = \varnothing$ for $i \neq j$.

We define a calculus of binary relations as a fragment of the previously defined algebra as follows.

$$\alpha ::= \top \mid M_i? \mid M_a \mid Z_j \mid \alpha + \alpha \mid \alpha \circ \alpha \mid \alpha - \alpha \mid \pi_\nu(\alpha) \mid \sigma_\Theta(\alpha) \mid \sim \alpha \mid \mu Z_j.\alpha \tag{7}$$

Here, $\top$ is a symbol that represents the tautology binary relation, M_i are propositions, M_a are actions, $\sim$ is a unary operation introduced in [41] which is a special kind of negation, as is used in modal temporal logics. Variables Z_j range over actions. We require that Z occurs positively (under an even number of negations $\sim$) in $\mu Z_j.\alpha$.

The calculus (7) is a fragment of our algebra (3) because $\sim$ is definable, as will be shown below; concatenation is product with a specified direction; and $\mu Z_j.\alpha$ is a version of the general recursion in (3) where the direction of fixed point computation is aligned with the direction of transitions, as will be seen from the semantics. (We limit ourselves to intuitions and do not give any precise propositions or theorems here, since it is only a project description paper). The operations we introduce in (7) subsume those we used in previous work [19, 20, 21, 22, 23, 24, 25]. In particular, concatenation here is exactly like sequential composition we used before, and feedback is definable using selection, where an input is equated with an output. There is a strong connection of the dynamic version of the algebra to the recent work [41], which we will explain in a future paper.

Semantics

Define a transition system $\mathcal{T} := (V; (M_a^{\mathcal{T}})_a, (M?_i^{\mathcal{T}})_i)$ that has domain V which is the class of all τ-structures, and it interprets all actions M_a as subsets of $V \times V$ denoted by $[\![M_a]\!]^{\mathcal{T}}$ and all monadic propositions $M_i?$ by structures (now nodes in the transition graph) $[\![M_i?]\!]^{\mathcal{T}} \subseteq V$ on which they are true. Variables Z_j that occur free in α are interpreted, as actions, as subsets of $V \times V$. Their interpretations are denoted $Z_j^{\mathcal{T}}$.

We define the extension $[\![\alpha]\!]^{\mathcal{T}}$ of formula α in $\mathcal{T}$ inductively as follows.

$[\![\top]\!]^{\mathcal{T}} := \{(\mathcal{B}, \mathcal{C}) \in V^{\mathcal{T}} \times V^{\mathcal{T}} \text{ for all } \mathcal{B}, \mathcal{C} \text{ in } V\}$.
$[\![M_i?]\!]^{\mathcal{T}} := \{(\mathcal{B}, \mathcal{B}) \in V^{\mathcal{T}} \times V^{\mathcal{T}} \mid \mathcal{B} \models_{M_i} M_i\}$.
$[\![M_a]\!]^{\mathcal{T}} := \{(\mathcal{B}_1, \mathcal{B}_2) \in V^{\mathcal{T}} \times V^{\mathcal{T}} \mid \mathcal{B}_1|_{\tau \setminus \varepsilon_{M_a}} = \mathcal{B}_2|_{\tau \setminus \varepsilon_{M_a}} \text{ and } \mathcal{B}_2 \models_{M_a} M_a\}$.
$[\![\alpha_1 + \alpha_2]\!]^{\mathcal{T}} := [\![\alpha_1]\!]^{\mathcal{T}} \cup [\![\alpha_2]\!]^{\mathcal{T}}$.
$[\![\alpha_1 \circ \alpha_2]\!]^{\mathcal{T}} := \{(\mathcal{A}, \mathcal{B}) \in V^{\mathcal{T}} \times V^{\mathcal{T}} \mid \exists \mathcal{C}((\mathcal{A}, \mathcal{C}) \in [\![\alpha_1]\!]^{\mathcal{T}} \text{ and } (\mathcal{C}, \mathcal{B}) \in [\![\alpha_2]\!]^{\mathcal{T}})\}$.
$[\![\alpha_1 - \alpha_2]\!]^{\mathcal{T}} := [\![\alpha_1]\!]^{\mathcal{T}} - [\![\alpha_2]\!]^{\mathcal{T}}$.
$[\![\pi_\nu(\alpha)]\!]^{\mathcal{T}} := \{(\mathcal{B}_1, \mathcal{B}_2) \in V^{\mathcal{T}} \times V^{\mathcal{T}} \mid \exists \mathcal{C}_1 \exists \mathcal{C}_2 \ ((\mathcal{C}_1, \mathcal{C}_2) \in [\![\alpha]\!]^{\mathcal{T}}, \mathcal{C}_1|_\nu = \mathcal{B}_1|_\nu \text{ and } \mathcal{C}_2|_\nu = \mathcal{B}_2|_\nu)\}$.
$[\![\sim \alpha]\!]^{\mathcal{T}} = \{(\mathcal{B}, \mathcal{B}) \in V^{\mathcal{T}} \times V^{\mathcal{T}} \mid \forall \mathcal{B}' \ (\mathcal{B}, \mathcal{B}') \notin [\![\alpha]\!]^{\mathcal{T}}\}$, no outgoing α-transition.
$[\![\mu Z_j.\alpha]\!]^{\mathcal{T}} := \bigcap \{R \subseteq V^{\mathcal{T}} \times V^{\mathcal{T}} \ : \ [\![\alpha]\!]^{\mathcal{T}[Z:=R]} \subseteq R\}$.

Semantics of selection requires a recursive definition. Case 3 for $\sigma_{(P \equiv Q)}$ coincides with Feedback operator used in our previous work.

$$[\![\sigma_{(P \equiv Q)}(\alpha)]\!]^{\mathcal{T}} :=$$

$$\left\{ (\mathcal{B}_1, \mathcal{B}_2) \in V^{\mathcal{T}} \times V^{\mathcal{T}} \;\middle|\; \begin{array}{l} \text{Case 1: } (\mathcal{B}_1, \mathcal{B}_2) \in [\![\alpha]\!]^{\mathcal{T}} \text{ and } vocab(\Theta) \subseteq \sigma_\alpha \text{ and } \mathcal{B}_1 \models_{\text{FO}} \Theta \text{ or} \\ \text{Case 2: } (\mathcal{B}_1, \mathcal{B}_2) \in [\![\alpha]\!]^{\mathcal{T}} \text{ and } vocab(\Theta) \subseteq \varepsilon_\alpha \text{ and } \mathcal{B}_2 \models_{\text{FO}} \Theta \text{ or} \\ \text{Case 3: } P \in \sigma_\alpha \text{ and } Q \in \varepsilon_\alpha \text{ and} \\ \exists \mathcal{C} \ ((\mathcal{C}, \mathcal{B}_2) \in [\![\alpha]\!]^{\mathcal{T}} \text{ and } \mathcal{B}_1|_{\tau \setminus \{P\}} = \mathcal{C}|_{\tau \setminus \{P\}}, \mathcal{B}_2 \models_{\text{FO}} (P \equiv Q)) \end{array} \right\}.$$

$$[\![\sigma_{(P\neq Q)}(\alpha)]\!]^{\mathcal{T}} := \left\{ (\mathcal{B}_1, \mathcal{B}_2) \in V^{\mathcal{T}} \times V^{\mathcal{T}} \;\middle|\; \begin{array}{l} \text{Case 1: same} \\ \text{Case 2: same} \\ \text{Case 3: } P \in \sigma_\alpha \text{ and } Q \in \varepsilon_\alpha \text{ and } P^{\mathcal{B}_1} \neq Q^{\mathcal{B}_2} \end{array} \right\}.$$

$$[\![\sigma_{(\Theta_1 \oplus \Theta_2)}(\alpha)]\!]^{\mathcal{T}} := \left\{ (\mathcal{B}_1, \mathcal{B}_2) \in V^{\mathcal{T}} \times V^{\mathcal{T}} \;\middle|\; (\mathcal{B}_1, \mathcal{B}_2) \in [\![\sigma_{\Theta_1}(\alpha)]\!]^{\mathcal{T}} \oplus (\mathcal{B}_1, \mathcal{B}_2) \in [\![\sigma_{\Theta_2}(\alpha)]\!]^{\mathcal{T}} \right\},$$

where $\oplus$ stands for either $\wedge$ (and) or $\vee$ (or).

Note that the semantics of $\mu Z_j.\alpha$ is exactly like that of the least fixed point operator in the modal mu-calculus $L\mu$. One direction of our project is to investigate the precise connections of various fragments of our algebra with $L\mu$ and other related logics.

Some definable operations are:

$$\begin{array}{l} D := \sigma_{\bigwedge_{P,Q\in\tau} P\equiv Q} \top(P,Q), \quad \text{the diagonal relation,} \\ \bot := \top - \top. \end{array}$$

By these definitions,

$$\begin{array}{l} [\![D]\!]^{\mathcal{T}} = \{(\mathcal{B},\mathcal{B}) \in V^{\mathcal{T}} \times V^{\mathcal{T}}\}, \\ [\![\bot]\!]^{\mathcal{T}} = \varnothing. \end{array}$$

Let π_1 abbreviates projection onto the first argument of the binary relation (onto all the inputs). Thus,

$$[\![\pi_1(\alpha)]\!]^{\mathcal{T}} := \{(\mathcal{B},\mathcal{B}) \in V^{\mathcal{T}} \times V^{\mathcal{T}} \mid \exists \mathcal{B}' \;\; (\mathcal{B},\mathcal{B}') \in [\![\alpha]\!]^{\mathcal{T}}\}.$$

This operation identifies the states in V where α "holds".

We have several important equalities between algebraic expressions (some are from [41]).

$$\begin{array}{l} M_1? \circ M_2? \;=\; M_1? \times M_2?, \\ \sim \alpha \;=\; (D - \pi_1(\alpha)), \\ \pi_1(\alpha) \;= \sim\sim \alpha, \\ D \;= \sim \bot. \end{array}$$

Example 8. *This modular system can be used by a company that provides logistics services. It decides how to pack goods and deliver them. It solves two NP-complete tasks interactively, – Multiple Knapsack (module M_K) and Travelling Salesman Problem (module M_{TSP}). The system takes orders from customers (items $Items(i)$ to deliver, their profits $p(i)$, weights $w(i)$), and the capasity of trucks available $c(t)$, decides how to pack $Pack(i,t)$ items in trucks, and for each truck, solves a TSP problem. The feedback about solvability of TSP is sent back to M_K. Module M_{TSP} takes a candidate solution from M_K, together with the graph of cities and routes with distances, allowable distance limit and destinations for each product. The output of this module is the route, for each truck $Route(t,n,c)$, where t is a truck, n is the number in the sequence, and c is a city. The Knapsack problem is written, in, e.g. Integer Linear Programming (ILP), and TSP in Answer Set Programming (ASP). The modules M_K and M_{TSP} are composed in sequence, with a feedback going from an output of M_{TSP} to an input of M_K. A solution to the compound module, M_{LSP}, to be acceptable, must satisfy both sub-systems.*

Model Expansion and Model Checking Tasks for Modular Systems

The model expansion task for atomic modules introduced above generalizes to modular systems. The goal here is, given input relations, produce structures that satisfy the entire algebraic expression α.

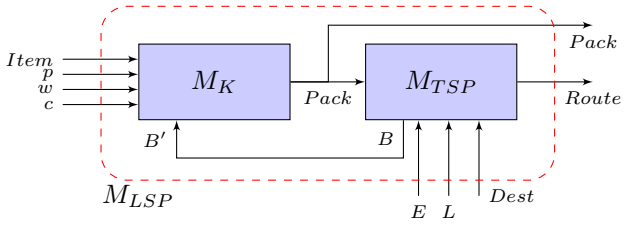

Figure 2: Logistics Service Provider $\sigma_{B \equiv B'}(M_K \circ M_{TSP})$.

Given: $\mathcal{B}|_\sigma$ and algebraic expression E with input symbols σ. Find: $\mathcal{B}$ such that $\mathcal{B}$ satisfies α. Structure $\mathcal{B}$ expands structure $\mathcal{B}|_\sigma$ and is called a *solution* for modular system represented by E for particular input $\mathcal{B}|_\sigma$. This task generalizes Model Expansion task introduced in [40] to the case of multi-language constraint solving.

A very naive method to solve model expansion for a modular system α would be to guess a structure expanding the input, and to check if it satisfies the algebraic expression. However, one can also develop an algorithm that identifies the set of all states $S \subseteq V$ in the transition system where an algebraic expression holds. The states in S will contain all expansions, for all instances. Then one can check whether a particular instance structure is in that set. A basic way to obtain S is by labelling the states of $\mathcal{T}$ by sub-expressions of α that hold in those states, going bottom-up on the structure of α. A better way is to use Binary Decision Diagrams (BDDs) and perform this labelling symbolically, as is standard in symbolic model checking. Notice that model expansion in the "flat" setting becomes model checking in the "modal" setting.

Expressive Power of the Algebraic Operations (Data Complexity)

To talk about solvability of modular systems, we need to discuss the expressive power of the formalism. In [26], we study the version of algebra with information flow, with operations of Sequential Composition, Projection, Feedback and Union only.[14] We study data complexity of the MX task, in terms of the size of the structure given on the input. In Codd's relational algebra, all operations are of linear time complexity. Here, there are *power-preserving* and *power-increasing* operations. The increase is due to added guessing power. Sequential composition and complementation are power-preserving. Union is power-increasing when there is an intersection of the output vocabularies (because union acts as an OR). Projection adds power when it hides inputs, they become expansions. Feedback equates an instance symbol to an expansion one. This operation adds guesses because a former input is no longer being "read" in the input structure, its interpretation is guessed and checked to be be equal to the extension of the expansion predicate, which is determined by the internal laws of the module. Note that the $\sim$ operation moves the complexity of MX task to a higher level of the polynomial time hierarchy because it contains an implicit universal second-order quantifier.

Finite model theory (see [30]) connects model theory and database research with computational complexity. Since [40], we have been arguing for the importance of capturing complexity classes in KR formalisms aiming at practical applications. Such capturing results show that all problems (and no more) in a complexity class are expressible. The "no more" part is responsible for implementability. In a simple fragment, we look at Sequential Composition, Projection, Feedback and Union. Capturing NP (on structures with finite active domain) holds for all power-increasing operations (in the simple fragment we considered), if individual modules are in P. More generally, since atomic modules can be of arbitrary complexity, "NP-step" is added by a modular system, in the fragment we mentioned, compared to the highest power among atomic modules [26]. The fundamental importance of this fact is that *a solver for this fragment,*[15] *working in cooperation with solvers of individual modules, is* **implementable**

[14] The complexity of the other operations will be explained in a future paper.

[15] In fact, implementability holds for a larger fragment than we studied in [26]. At that time, the algebra had fewer operations.

with the current solver technology that is able to handle NP.

Solving Modular Model Expansion (Meta-Solving)

Solutions to modular systems can be computed by iteratively questioning individual modules with partial 3-valued structures. If a structure is accepted by a module, the module may suggest (through a formula) what other properties should be true. If a structure is rejected by a module, a reason (also a formula) may be given. The algorithm reasons with accumulated formulas, using partial structures as a data structure, and may retract its earlier decisions. Information is added until all unknowns are filled. A CDCL-like algorithm for solving modular systems is given in a joint work with David Mitchell [24]. A high-level algorithm that formulated the idea behind meta-solving, where each module is viewed as an oracle that accepts or rejects partial structures and returns reasons and advice, is given in FROCOS 2011 paper with Shahab Tasharrofi [18] and in his PhD thesis. Shahab also proposed the term "meta-solving" in a recent discussion. Instantiations of the algorithm [18] were shown to capture the work of SMT (DPPL(T)), ASP-CP, IDP, ILP (represented as modular systems) in [46], and in Xiongnan (Newman) Wu's thesis [20]. An important future research direction is to explain good algorithmic properties of various fragments of the dynamic version of the algebra, and to explore those properties in algorithms for meta-solving.

Algebra of Modular Systems as Process Algebra

Process Algebras are mathematical languages with well-defined semantics for describing and verifying properties of concurrent communicating systems. They originated in the work of Milner [47, 48, 49] and Hoare [50, 51], and then generated enormous literature. Process Algebras are closely connected to Petri nets, automata theory and formal languages. In process algebras, complex programs, often concurrent, are specified by algebraic terms. For example, term $((a + b) \circ c \circ b) \circ ((e||d) \circ c)$ represents actions connected by sequential ($\circ$) and parallel ($||$) compositions and choice ($+$). It is very interesting that *the same algebra, our algebra of modular systems can be viewed as a process algebra.* E.g., a particular type of parallel composition is expressible through Product where vocabularies are distinct, selection can be used for synchronization and communication.

Behavioural Equivalences, Bisimulation

The notion of equivalence (and containment) of modular systems above is model-theoretic and does not take into account behaviour of modular systems over time. Several notions of congruence are introduced in the process algebras research community, and **bisimulation** (and simulation) is among the most important ones. Intuitively, two states v and v' in transition systems $\mathcal{T}$ and $\mathcal{T}'$ are bisimilar (observationally equivalent) if they have the same local properties and any transition from v to w in $\mathcal{T}$ must be matched by a transition v' to w' of the same kind in $\mathcal{T}'$ (and vice versa) so that w and w' are again bisimilar. Bisimilar systems perform the same sequences of actions, and after each sequence exhibit, recursively, the same behaviour. Modular systems that are equivalent model-theoretically, are not necessarily bisimilar. For example, while $(M_1 + M_2) \times M_3 = (M_1 \times M_3) + (M_2 \times M_3)$, for any choice of actions M_1, M_2, M_3, these two compound modules are not bisimilar. Study of bisimulation and other behavioural equivalences for our formalism is a future research direction.

Connections to Other Logics

There are very interesting connections of our algebra of modular systems to other logics. In particular, Bisimulation-Safe Fixed Point Logic (BSFP) recently introduced by F. Abu Zarid, E. Grädel and S. Jaax

[41] is a fragment of the logic of hybrid model expansion explained above. Also, Propositional Dynamic Logic is also a fragment. A connection to the situation calculus and Golog [35, 45] is also pretty clear.

4 Discussion

More on Closely Related Work

To our knowledge, no other work proposed a general framework for *high-level control* for programming from reusable components in knowledge-intensive computing, with the desiderata as in the Introduction, and suggested both model-theoretic and operational (based on a transition system) views.

While the emphasis in most of our earlier work on Modular Systems (starting from [18]) was on the Model Expansion task, *the notion of a module in [18] is mathematically the same as in the current paper*. A module is a *class of structures*. This is stated explicitly in our NMR'14 paper [22]. This is because the "yes" instances for the MX task, expanded with solutions, constitute classes of structures. For example, the module of 3-Colouring is a class of structures that are graphs expanded with proper 3-colourings. As in the curent paper, modules in [18] and our subsequent papers were given by decision procedures of arbitrary complexity, could be axiomatized in, e.g., ASP, ILP, or SMT theories, and the input and the output vocabularies were specified. Thus, we have been mostly studying the "dynamic" (or "modal") fragment of the general algebra. We consider the dynamic version very important because it formalizes a typical task in hybrid multi-language constraint solving, and has good computational properties. Modular systems representing SMT, branch-and-cut based ILP, ASP-CP combinations are given in [20, 21] to illustrate a high-level algorithm [18] for solving modular systems (what we now call meta-solving).

We have shown the equivalence of model-theoretic (given via a class of structures) and operational (given via a transition system) semantics of Modular Systems in our NMR'14 paper [22].

Information propagation in Modular Systems, as defined in [18], happened through equivalent vocabulary symbols. In the current version, selection operator allows us to connect modules with different vocabularies.

In our earlier papers, we had the requirements, for some algebraic operations, of modules being composable (no output interference) and independent (no cyclic dependencies). The requirements are inherited from [52]. However, even in the earlier versions of the algebra, composability was redundant since the semantics is defined so that when there is an interference, there is no model. This requirement is dropped now. With the introduction of the selection operator, independence is no longer needed because modules no longer need to have equal vocabulary symbols in order to connect.

There are important works by other researchers that study related issues. Recently, [53] introduced a formalism with compositions (essentially, conjunctions) of modules given through solver-level inferences. The importance of that work is in a unifying mathematical view on the work of a variety of solvers and their combinations, which we believe is essential. In NMR'14 paper [22], we generalized inference-based modules of [53] as another representation of modules of our formalism of Modular Systems. That generalization represents inference modules as classes of structures, as introduced in [18].

The basic notions of [54], i.e., the notion of a module and information propagation through the same vocabulary symbols, are identical, in its mathematical essence, to those in our work [18, 22][16]. Composition in that framework is the same as Product here. Compositions in [54] cannot achieve the same expressive power as our algebra since other basic algebraic operations are not expressible through Product. A large part of [54] is devoted to establishing connections to Multi-Context Systems (MCSs) [34]. MCSs combine knowledge bases in arbitrary languages, under arbitrary semantics (that do not have to

[16] Modules in [54] are axiomatized (represented by theories) in logics associated with modules.

be model-theoretic) through rules of logic programming with negation as failure. The authors of [54] give a translation from MCSs to modular systems with one operations (Product). They collect all rules leading to a module and add them to that module. The semantic of the module is modified so that, when limited to the original module, it is the same as before, and when limited to the rules, it is stable model semantics. They conclude that "we showed that modular systems, despite their simplicity, are as expressive as multi-context systems". This claim is somewhat misleading. The expressive powers of different formalisms are usually compared *in terms of the same units*. When some entities are given (modules in this case), a comparison is made with respect to what new classes of structures the formalisms can axiomatize using those given entities, or what relations they can define with respect to those entities. Making individual modules stronger in *one* of the compared formalisms does compensate for the weakness of the operations, but does not imply the equivalence of the expressive powers. It is pretty clear that, given a set of modules, the class of structures one can define using, as links between modules, rules with negation as failure under stable model semantics, as in MCSs, is significantly larger than the class of structures definable using conjunctions only, as in the formalism of [54]. When membership in each atomic module is polytime, membership in compound modules that are produced using conjunctions only will be polytime as well, while membership in MCSs would be NP-hard in general. Thus, MCSs are *significantly* more expressive than the formalism of [54]. We also studied connections of a generalization of our algebra with MCS in our KR'2014 paper [23]. As a side effect, and to generalize MCSs, we defined a generalization of our algebra under *supported model semantics*.

The paper that originally inspired our Modular Systems framework is [52], but we developed a model-theoretic approach and provided additional operations (in [52], projections and sequential compositions are possible, but not feedbacks or the other operations used here.).

A very interesting related work is by D. Fontaine, L. Michel, P. Van Hentenryck [55]. The authors develop a framework where, what they call "model combinators" are used to compose models, i.e., descriptions in some modelling language. The intention of model combinators is similar to our algebraic operations, but there is no basic combinators, combinators are introduced for each application. What they call "runnables" are essentially our model expansion tasks, but they do not use model theory. The authors also introduce model hierarchies, which track relationships among models (which are high-level descriptions in their, and CP in general, terminology). The framework has an implementation, and it would be interesting to connect it to mathematical foundations we are developing.

An important and active direction already mentioned above is Satisfiability Modulo Theory (SMT), (see [7]). Efficient solvers have been developed, e.g. Z3 [56], and its extension with recursion [57]. SMT can be viewed as a modular system with a SAT module, and where some modules are axiomatized by built-in theories of SMT solvers. Our work is a generalization of this work to *arbitrary* modules, not necessarily SAT or represented by built-in theories, with an algebra to represent sophisticated connections between modules. Similarly to SMT, combined solvers such as [4] also introduce combinations of specific formalisms, for example ASP and CP.

Many combinations of logical formalisms have been proposed, notably [58, 59]. However, we need combinations where logics are not embedded into another logic, but used 'as is'.

There are many interesting directions for future work, among them an algebraic formalization of lazy model expansion (on-demand grounding) [15] and a study of a version of our algebra where user's inputs are allowed during the execution. However, in that latter direction, it is very easy to obtain a formalism with undecidable basic properties, so careful constructions need to be used. Many more potential directions and results are listed in the body of the paper.

Conclusion

Integrating different constraint solving technologies is a difficult task, and it is a cornerstone of current trends in several constraint solving communities. The main challenge is that the programs may be written in different languages (even legacy languages), and rely on different solving technologies. We identified a basis for integration, defined what individual modules are. We developed an algebra for combining modules. In this algebra, we identify *five basic operations*, through which other useful operations are expressible, and we introduce Recursion. Identifying a small set of basic operators is very important both for proving theoretical results and for progress in practical technology. We show that other useful operations are expressible using the basic operations. We discuss two versions of the algebra, the general one and the one with information flow. We briefly discuss the complexity of the formalism, and the descriptive complexity of modular systems, as a function of expressiveness of individual modules. We also discuss equivalence and inclusion tasks for modular systems. Our formalization leads to several research directions, many of which are described in the body of the paper.

We believe that our formalism will help developing practical modelling tools for rapid prototyping and hierarchical modelling in constraint solving, for specifying high-level solving control and for effective program reuse.

Acknowledgements

I am grateful to my PhD and MSc students and co-authors who participated in this project. Discussions with Andrei Bulatov, Giuseppe De Giacomo, Marc Denecker, Yves Lesperance, Mirek Truszczynski and David Mitchell at various stages of this projects were very helpful. I am also grateful to the anonymous referees who's comments helped to improve the paper, and to Bart Bogaerts for his comments to an earlier version. This research is supported by Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Ann. Math. Artif. Intell. **53**(1-4) (2008) 251–287

[2] Balduccini, M., Lierler, Y., Schüller, P.: Prolog and ASP inference under one roof. In Cabalar, P., Son, T.C., eds.: Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings. Volume 8148 of Lecture Notes in Computer Science., Springer (2013) 148–160

[3] Hurley, B., Kotthoff, L., Malitsky, Y., O'Sullivan, B.: Proteus: A hierarchical portfolio of solvers and transformations. In Simonis, H., ed.: Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings. Volume 8451 of Lecture Notes in Computer Science., Springer (2014) 301–317

[4] Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: Proceedings of the 25th International Conference on Logic Programming (ICLP'09). Lecture Notes in Computer Science (LNCS), Springer-Verlag (2009) 235–249

[5] Zhou, N.F., Fruhman, J.: A user's guide to Picat (version 1.2) (2013-2015) Picat programming language website: `http://www.picat-lang.org/`.

[6] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). J. ACM **53**(6) (2006) 937–977

[7] Sebastiani, R.: Lazy satisfiability modulo theories. Journal of Satisfiability, Boolean Modeling and Computation (JSAT) **3** (2007) 141–224

[8] Amir, E., McIlraith, S.A.: Partition-based logical reasoning for first-order and propositional theories. Artif. Intell. **162**(1-2) (2005) 49–88

[9] Abseher, M., Bliem, B., Charwat, G., Dusberger, F., Hecher, M., Woltran, S.: The D-FLAT system for dynamic programming on tree decompositions. In Fermé, E., Leite, J., eds.: Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings. Volume 8761 of Lecture Notes in Computer Science., Springer (2014) 558–572

[10] Charwat, G., Woltran, S.: Efficient problem solving on tree decompositions using binary decision diagrams. In Francesco Calimeri, Giovambattista Ianni, M.T., ed.: Logic Programming and Nonmonotonic Reasoning, 13th International Conference, LPNMR 2015, Lexington, September 27-30, 2015. Proceedings. Lecture Notes in Computer Science, Springer (2015)

[11] Gebser, M., Janhunen, T., Rintanen, J.: Answer set programming as SAT modulo acyclicity. In Schaub, T., Friedrich, G., O'Sullivan, B., eds.: ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014). Volume 263 of Frontiers in Artificial Intelligence and Applications., IOS Press (2014) 351–356

[12] Gange, G., Stuckey, P.J., Lagoon, V.: Fast set bounds propagation using a BDD-SAT hybrid. J. Artif. Intell. Res. (JAIR) **38** (2010) 307–338

[13] Tasharrofi, S., Ternovska, E.: Logical machinery of heuristics (preliminary report). In: Proceedings of the 4th Workshop on Logic and Search Heuristics (LaSH'15), co-located with VSL 2014: VIENNA SUMMER OF LOGIC 2014. (2014)

[14] Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: dlvhex: A prover for semantic-web reasoning under the answer-set semantics. In: 2006 IEEE / WIC / ACM International Conference on Web Intelligence (WI 2006), 18-22 December 2006, Hong Kong, China, IEEE Computer Society (2006) 1073–1074

[15] de Cat, B., Denecker, M., Bruynooghe, M., Stuckey, P.J.: Lazy model expansion: Interleaving grounding with search. J. Artif. Intell. Res. (JAIR) **52** (2015) 235–286

[16] Bogaerts, B., Jansen, J., De Cat, B., Janssens, G., Bruynooghe, M., Denecker, M.: Meta-level representations in the IDP knowledge base system: Towards bootstrapping inference engine development. In Mitchell, D., Denecker, M., eds.: International Workshop on Logic and Search (Lash 2014), Vienna, July 18, 2014. (2014)

[17] Francis, K.: Rethinking the quest for declarativity. In: ModRef'14, International Workshop on Constraint Modelling and Reformulation. (2014)

[18] Tasharrofi, S., Ternovska, E.: A semantic account for modularity in multi-language modelling of search problems. In: Proceedings of the 8th International Symposium on Frontiers of Combining Systems (FroCoS). (October 2011) 259–274

[19] Tasharrofi, S.: Modular Systems. PhD thesis, Simon Fraser University, Burnaby, BC, Canada (December 2013)

[20] Wu, X.N.: Solving model expansion tasks: System design and modularity. Master's thesis, Simon Fraser University, Burnaby, BC, Canada (August 2012)

[21] Tasharrofi, S., Wu, X.N., Ternovska, E.: Solving modular model expansion: Case studies. In: Postproceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management and 25th Workshop on Logic Programming, Lecture Notes in Artificial Intelligence (LNAI) (2012) 175–187

[22] Tasharrofi, S., Ternovska, E.: Three semantics for modular systems. In: Proceedings of NMR'2014. (2014)

[23] Tasharrofi, S., Ternovska, E.: Generalized multi-context systems. In Baral, C., Giacomo, G.D., Eiter, T., eds.: Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014, AAAI Press (2014)

[24] Mitchell, D.G., Ternovska, E.: Clause-learning algorithms for modular systems. In Francesco Calimeri, Giovambattista Ianni, M.T., ed.: Logic Programming and Nonmonotonic Reasoning, 13th International Conference, LPNMR 2015, Lexington, September 27-30, 2015. Proceedings. Lecture Notes in Computer Science, Springer (2015)

[25] Ensan, A., Ternovska, E.: Modular systems with preferences. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI2015), Buenos-Aires, Argentina, AAAI Press (January 6–12 2015)

[26] Tasharrofi, S., Ternovska, E.: Modular systems. In: Proceedings of Workshop on Hybrid Reasoning at IJCAI'15. (2015)

[27] Wittocx, J., Marién, M., Denecker, M.: The IDP system: A model expansion system for an extension of classical logic. In: Proceedings of the 2nd Workshop on Logic and Search. (2008) 153–165

[28] Aavani, A., Wu, X.N., Tasharrofi, S., Ternovska, E., Mitchell, D.G.: Enfragmo: A system for modelling and solving search problems with logic. In: 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. (2012) 15–22

[29] Hölldobler, S., Schweizer, L.: Answer set programming and Clasp, a tutorial. In Hölldobler, S., Malikov, A., Wernhard, C., eds.: Proceedings of the Young Scientists' International Workshop on Trends in Information Processing (YSIP), CEUR Workshop Proceedings (2014) 77–95

[30] Libkin, L.: Elements of Finite Model Theory. Springer Verlag (2004)

[31] Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints **13** (2008) 268–306

[32] Opatrny, J.: Total ordering problem. SIAM J. Comput **8**(1) (1979) 111–114

[33] Gelfond, M., Lifschitz, V.: Logic programming. In Warren, D.H., Szeredi, P., eds.: Proceedings of the 6th International Conference on Logic Programming (ICLP). MIT Press, Cambridge, MA, USA (1990) 579–597

[34] Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI'07) - Volume 1, AAAI Press (2007) 385–390

[35] Levesque, H., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.: GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming **31** (1997) 59–84

[36] Chandra, A.K., Merlin, P.M.: Optimal implementation of conjunctive queries in relational data bases. In Hopcroft, J.E., Friedman, E.P., Harrison, M.A., eds.: Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA, ACM (1977) 77–90

[37] Grädel, E., Kolaitis, P.G., Libkin, L., Marx, M., Spencer, J., Vardi, M., Venema, Y., Weinstein, S.: Finite Model Theory and Applications. Springer (2007)

[38] Denecker, M., Ternovska, E.: A logic of non-monotone inductive definitions. ACM transactions on computational logic (TOCL) **9**(2) (2008) 1–51

[39] Eiter, T., Gottlob, G., Mannila, H.: Expressive power and complexity of disjunctive datalog under the stable model semantics. In von Luck, K., Marburger, H., eds.: Management and Processing of Complex Data Structures, Third Workshop on Information Systems and Artificial Intelligence, Hamburg, Germany, February 28 - March 2, 1994, Proceedings. Volume 777 of Lecture Notes in Computer Science., Springer (1994) 83–103

[40] Mitchell, D.G., Ternovska, E.: A framework for representing and solving NP search problems. In: Proc. AAAI'05. (2005) 430–435

[41] Abu Zaid , F., Grädel, E., Jaax, S.: Bisimulation safe fixed point logic. In Goré, R., Kooi, B.P., Kurucz, A., eds.: Advances in Modal Logic 10, invited and contributed papers from the tenth conference on "Advances in Modal Logic," held in Groningen, The Netherlands, August 5-8, 2014, College Publications (2014) 1–15

[42] Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artificial Intelligence **187-188** (August 2012) 52–89

[43] Mitchell, D.G., Ternovska, E.: Expressiveness and abstraction in ESSENCE. Constraints **13(2)** (2008) 343–384

[44] Lierler, Y., Truszczynski, M.: Transition systems for model generators - A unifying approach. TPLP **11**(4-5) (2011) 629–646

[45] Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press (2001)

[46] Tasharrofi, S., Wu, X.N., Ternovska, E.: Solving modular model expansion: Case studies. In Tompits, H., Abreu, S., Oetsch, J., Pührer, J., Seipel, D., Umeda, M., Wolf, A., eds.: Applications of Declarative Programming and Knowledge Management - 19th International Conference, INAP 2011, and 25th Workshop on Logic Programming, WLP 2011, Vienna, Austria, September 28-30, 2011, Revised Selected Papers. Volume 7773 of Lecture Notes in Computer Science., Springer (2011) 215–236

[47] Milner, R.: A Calculus of Communicating Systems. Volume 92 of Lecture Notes in Computer Science.

Springer (1980)

[48] Milner, R.: Synthesis of communicating behaviour. In Winkowski, J., ed.: Mathematical Foundations of Computer Science 1978, Proceedings, 7th Symposium, Zakopane, Poland, September 4-8, 1978. Volume 64 of Lecture Notes in Computer Science., Springer (1978) 71–83

[49] Milner, R.: A modal characterisation of observable machine-behaviour. In Astesiano, E., Böhm, C., eds.: CAAP '81, Trees in Algebra and Programming, 6th Colloquium, Genoa, Italy, March 5-7, 1981, Proceedings. Volume 112 of Lecture Notes in Computer Science., Springer (1981) 25–34

[50] Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10) (October 1969) 576–580

[51] Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8) (August 1978) 666–677

[52] Järvisalo, M., Oikarinen, E., Janhunen, T., Niemelä, I.: A module-based framework for multi-language constraint modeling. In: Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09). Volume 5753 of Lecture Notes in Computer Science (LNCS)., Springer-Verlag (2009) 155–168

[53] Lierler, Y., Truszczynski, M.: Abstract modular inference systems and solvers. In: Proceedings of the 16th International Symposium on Practical Aspects of Declarative Languages (PADL'14). (2014)

[54] Lierler, Y., Truszczyński, M.: An abstract view on modularity in knowledge representation. In: Proceedings of the 27th AAAI Conference on Artificial Intelligence. (2015)

[55] Fontaine, D., Michel, L., Hentenryck, P.V.: Model combinators for hybrid optimization. In Schulte, C., ed.: Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings. Volume 8124 of Lecture Notes in Computer Science., Springer (2013) 299–314

[56] de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In Ramakrishnan, C.R., Rehof, J., eds.: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Volume 4963 of Lecture Notes in Computer Science., Springer (2008) 337–340

[57] Hoder, K., Bjørner, N., de Moura, L.M.: μZ- an efficient engine for fixed points with constraints. In Gopalakrishnan, G., Qadeer, S., eds.: Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Volume 6806 of Lecture Notes in Computer Science., Springer (2011) 457–462

[58] Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. Artificial Intelligence **172** (2008) 1495 – 1539

[59] de Bruijn, J., Eiter, T., Polleres, A., Tompits, H.: Embedding non-ground logic programs into autoepistemic logic for knowledge-base combination. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI2007), Hyderabad, India, AAAI Press (January 6–12 2007)