Modeling the User's Constraint Domain

Georg Ringwelski

4C, UCC, Cork, Ireland g.ringwelski@4c.ucc.ie

Abstract. The availability of two decades of research in off-the-shelf constraint solvers makes Constraint Programming the method of choice in many application areas of Artificial Intelligence. The domain of these systems is naturally very abstract such that an elaborate modeling of the application domain in the constraint domain has to be made. In interactive applications, the found model is normally too abstract to be handled by the user. I propose a modeling schema that is oriented to the tackled application domain and still uses the powerful abstract solvers. I think this will allow for better software ergonomics as an Object-Oriented model is closer to the users expertise than a model in clp(FD) for example. The main idea is to move the interpretation work of user-input and solver-output from the application to the implementation of domain specific constraints.

1 Motivation

Constraint Programming (CP) has proven to be a very effective means to solve problems of manifold structures. Constraint Satisfaction or Optimization Problems (CSP/COP) are often used very successfully to model and solve real-world or benchmark problems with CP. The basic idea is to formulate problems in terms of variables, their respective domains and constraints and use a reusable constraint solver to find solutions to the thus modeled problems. Much research and software development has been made to successfully improve constraint solvers wrt. their capabilities and efficiency in the last two decades. This knowledge has been applied successfully to arbitrary problems in many academic and commercial applications today.

The development of these applications required in most cases a CP expert who modeled the actual problem in the constraint domain that is supported by the used solver. The most popular domains are finite intervals of integers (CHIP,clp(FD),ILOG Solver,ECL^{i}PS^e,SICStus), intervals of reals (clp($\mathfrak{R}$), ECL^{i}PS^e, SICStus) or finite enumerations of arbitrary numbers of values (JCL, ECL^{i}PS^e). The modeling effort is generally considered the main work of developing constraint-based applications. After the model was defined, the targeted input can be transformed into variables and constraints of the solver's domain.

This work was funded by the Embark initiative of the Irish Research Council for Science, Engineering and Technology under grant PD/2002/21. The Cork Constraint Computation Center is supported from Science Foundation Ireland under Grant 00/PI.1/C075.

In scheduling for example, tasks are often modeled as quadruples of numbers of fd-variables such that arithmetic constraints can be used to state dependencies between the tasks. For many application areas powerful, global constraints were developed to increase the performance of the solver. These constraints also have a declarative semantics in the solver's domain, e.g. the algebra $\mathbb{Z}$ in clp(FD). The `cumulative` constraint over quadruples of fd-variables or integers in CHIP for example is defined as a set of in-equations over the input variables and values. The used solver can then solve the modeled problem, e.g. find an assignment of constraint variables to integers such that all arithmetic constraints are satisfied. After that this solution must be interpreted along the lines of the chosen model to produce the desired output. The intended output is in most cases not expressible in the domain of the constraint solver, but in some application domain like work-flow schedules for example. Consequently the solver's output has to undergo an interpretation in the application domain. In all the thus designed constraint-based applications the decision for the use of a particular solver will force the programmer to use the abstract modeling domain that is supported by the solver. As I observed in many projects, the application will then be developed somehow around the chosen solver and is not designed according to a Software Engineering (SWE) method that targets good interactivity. In the SWE literature (e.g. [14]) it is commonly observed that the chosen model should be as close to the real-world as possible in order to improve among general software quality features also software ergonomics.

It has often been discussed in the constraints community, that users are not able to provide a sufficient amount of domain knowledge to enable systems to find the desired solutions. Consequently there is current research on knowledge acquisition [12], which is to provide a way to get just that kind of input from the user, which is needed to solve her problem with the means of CP. One main aspect of this work is to "ask the right questions". It is generally agreed that the question "Do you want a cumulative among these quadruples?" is not what users can answer and should consequently not be asked. Asking questions and using the answers requires a bi-directive translation of what the user understands and what the solver can handle. This translation can be made much easier with the use of Complex Constraint Domains. Using this technique, constraints can be designed according to the user's knowledge, such that on the one hand side the user will understand, what kind of input is requested and on the other hand the system is able to process this input adequately. With the proposed modeling framework we can still use off-the-shelf constraint solvers, but the software design does not have to be oriented to their constraint domain, but to the considered objects of the application domain the user understands. In this setting, the work of translation is moved from the application to the implementation of the constraints. Once the semantics of the application-oriented constraints is implemented by an algorithm that makes them processable by a solver, they can be used for arbitrary user interaction. I think, this will make user interaction more effective and thus improve the ergonomics of constraint-based software.

The proposed approach is language and (of course) solver independent. The paper is to provide a general SWE method to use constraints in complex domains. There are some standard requirements to the implementation platform, which are met by all modern programming languages. The constraint solver that is to be used has to provide a feature, which is supported in most of today's systems and is a widely accepted basis for CP. This is the possibility to implement constraints by means of propagators [13] or domain reduction functions. These are procedures that are executed whenever a constraint is triggered, i.e. when the solver detected, that the constraint might infer new knowledge in the current state.

2 Traditional Constraint Domains

Constraint Satisfaction Problems (CSP) are defined by a set of variables and a conjunction of constraints over these variables. Constraint Optimization Problems (COP) use an additional objective function which defines a metric on all solutions of the CSP. The declarative semantics of every constraint is usually given by some logic expression, which states restricting properties of the constraint's variables. In CP, this restriction is projected to the variables' respective domain. Domains are given as sets of allowed values for every variable and constraints can eliminate values from these sets, that are proven not to satisfy the constraint. Operationally, constraints can be given either explicitly or implicitly:

1. The explicit definition of a constraint is given by a relation over the Cartesian product of its variables' domains. Thus it explicitly enumerates all allowed combinations of values that will model its declarative semantics: Let $v_1, ..., v_n$ be variables with respective domains $D_1, ..., D_n$, then a constraint c over $v_a, v_b, ...$ is defined as $c \subseteq D_a \times D_b \times ....$
2. The implicit definition of a constraint is normally given by propagators[13]. These are procedures, that may restrict variable domains according to the properties of other variables. In this setting, c defines at least one propagator, which implies a domain reduction function[1] in $D_1 \times ... \times D_n$. The domain reduction has to be done in accordance to the intended semantics of the constraint.

It can easily be seen, that in both cases the domain of every variable is derived from a superset, or universe. This set, together with its typical operations and relations is referred to the **constraint domain** [8]. Or from the other point of view: A constraint domain is a single-sorted algebra with predicates (constraints) and the variable domains are subsets of its carrier. In all constraint solvers I am aware of, variable domains are subsets of some basic datatype that is supported by the host language. For example, the basic data type `integer` or `long` is used as the superset of all variable domains in constraint solvers for finite domains [2, 4, 10].

The constraint programmer can be expected to know the semantics of the basic operations and relations of this underlying datatype, since it is part of the

programming language she is using. This knowledge is used to make the constraints, which operate on constraint variables instead of values, understandable for her. For example, every programmer will understand the semantics of the constraint $X + Y < Z$ intuitively from her knowledge of the operation $+$ and the relation $<$ in $\mathbb{Z}$, although the used operations are not the ones, she knows and have other operational and denotational semantics. The idea of declarative programming with constraints is thus to overload these well known operations and implement their domain-counterpart in constraints.

Technically a constructor is used to create constraint variables out of a description of their initial domain (e.g. `in` or `::` in many CLP systems). Over the thus created objects (i.e. constraint variables) the basic operations and relations, i.e. the constraint, can be re-defined containing the intended semantics. This has to be done explicitly for every operation and relation and can be a very difficult task having many pitfalls [3]. There are two requirements to the re-definition of operations on constraint variables that have always to be met:

1. Whenever the variable's domain contains exactly one value, the constraints (i.e. relations and operations) have the same semantics as their underlying operations.
2. The domain reductions a constraint makes are always sound wrt. the declarative semantics. This means, that no values are pruned that cannot be proven not to be part of any solution.

Furthermore, constraints may cause the system to fail, if they detect, that no solution can be found for the specified problem. The second requirement implies the first one in the case of instantiated variables. In most cases this fact leads to an implementation of constraints with their underlying relation. This can also make sure, that the specification of both operations will match in this special case. This formal topic can be discussed in further work, for now the idea is demonstrated in the equation that specifies the `lt`-constraint in Example 1. In that example I also illustrate the idea of the construction of a constraint system upon a basic datatype. It shows part of a specification of a finite domain constraint system that is built upon some given integer type. I used the syntax of Algebraic Specification, which the reader can find a detailed description for in [5].

Example 1. Let be given a basic data type for integers with the successor operation and a less-than relation `lt`, which is modeled as boolean function:

```
INT =
 sorts: int, bool
 opns: zero: -> int
   true, false : -> bool
   succ: int -> int
   lt: int int -> bool
 eqns: lt(X,succ(X)) = true
   lt(succ(Y),zero) = false
   lt(X,succ(Y)) = lt(X,Y)
```

One can build a constraint system with the less-than constraint over finite domain variables upon this datatype by using `INT`, and a new type for variables, that contains as domain information the upper and lower bounds of the allowed interval of integers.

```
FD = INT +
 sorts: fd_var
 opns: mkVar: int int -> fd_var // constructor
   lt: fd_var fd_var -> bool // constraint
 eqns: lt(mkVar(X1,Y1),mkVar(X2,Y2)) = lt(X1,Y2)
```

The equation specifying the semantics of the `lt`-constraint says, that the constraint holds, whenever the lower bound of the first variable is less than the upper bound of the second. In this example, the operation `lt` is overloaded and used in both versions in the equation. The equation defines the semantics of `lt` on constraint variables to match the semantics of `lt` on integers whenever the domains contain exactly one value as stated in 1. on page 4.

In general, a constraint system is made up a data type for variables that specifies a set of allowed values and a set of constraints, which are relations over that data type.

Definition 1. *A constraint system for the domain D is given by a type t for constraint variables, a (total) function $dom : t \rightarrow \mathcal{P}(D)$ which assigns every variable its domain and a set of constraints which are boolean functions over arbitrary tuples of constraint variables.*

For the construction of a constraint system the basic datatype has to provide Booleans in order to allow the modeling of relations as boolean functions. Furthermore some structured set of values should be provided, that is to be used as superset for all variable domains. Given these basic prerequisites, a constraint system can be defined upon the basic datatype.

Proposition 1 *A constraint system can be constructed by:*

1. *A new type τ for constraint variables*
2. *An injective constructor $cs : \sigma_1 \times \sigma_2 ... \times \sigma_n \rightarrow \tau$ that creates a constraint variable from terms of type σ_i such that it uniquely defines a set of values, which is to become the variable's domain.*
3. *Boolean (constraint-)operations $c : \sigma_1 \times .... \times \tau \times ... \times \sigma_m \rightarrow bool$ that use at least one variable as input and are true, iff the constraint is satisfied for the input values, otherwise false.*

Any constraint system, given as a parameterized datatype with the proposed properties, can be generated by actualization with some basic datatype. The general module composition technique of actualization is described in detail in [6], where not only the forward directed construction of the syntax is described, but also an initial (in the category of all algebras with the specified syntax) algebra is constructed from the semantics of the basis and the parameterized data type.

3 Complex Constraint Domains

For benchmarking or illustration purposes, often mathematical problems or problems that use very compactly describable data are used. For most of such problems (many of them are collected at CSPlib[9]) no modeling of values is necessary, because the searched solutions consist of basic values such as numbers. The modeling effort, which is extensively discussed in the community for these problems concentrates on the identification of variables and constraints, such that some given constraint solver can handle the problem. The art or craft of modeling with constraints is often considered as to find variables, their respective domains and constraints, such that a problem can be solved fastest. This paper addresses a modeling issue that is often only marginally handled: The modeling of values. This task of modeling is not often considered in CP research, because it is obvious for the most of the mentioned problems or it is determined by the constraint solver that is to be used. However, as I want to integrate CP into standard Object-Oriented Software-Processes the constraint domains will in most cases have a structure that does not allow the use of standard types of variable domains and constraints. The constraints in this framework will normally restrict properties of objects that model real-world objects and will thus have a more complex internal structure.

Example 2. During the Object-Oriented Analysis for a meeting scheduling tool, meetings are likely to be modeled as objects. As I discussed in [11] the following could be one reasonable class for them:

Meeting			
Date	Time	Duration	Location
int day int month int year	int hour int minute	int minutes	string where

Constraints over such objects could be for example:

1. Meeting A must be before meeting B
2. No meetings overlap
3. Meetings should preferably be before lunch
4. Meeting A must not exceed 20 minutes

Just as in traditional CP, constraints can be hard (1,2,4) or soft (3), unary (4), binary (1) or global (2) and they can be user-defined (1,3,4) or inherent to the application domain(2).

As can be seen in the above example, in the Object-Oriented modeling with constraints it is adequate to use constraints over entities with a complex internal structure. The constraints restrict the properties of these structures wrt. one or more of their features. The superset of all these structures makes up the constraint domain of the respective application. The handling of such structured

constraint domains is the actual contribution of this paper. In traditional approaches, only flat datatypes are considered as constraint domains. I will show in this paper, that also complex constraint domains can be handled by traditional, re-usable constraint solvers, if the implementation of the complex structures provides some required functionality.

Following the CP approach, the variable domains are derived from the constraint domain. Thus, I assume that the constraint domain contains all possible entities that can be described by the available basic datatypes and the provided constraint systems. As I have stated earlier, a constructor has to exist for every type of constraint variable.

Definition 2. *Let be given a set of basic datatypes D with respective sets of sorts $sorts(D)$ and a set of constraint systems with the set of constructors CS. The set $Dom_{D,CS}$ of all* **complex constraint domains** *over D and CS is the closure of $Dom_{D,CS}$ wrt. the following implication: $\tau_1, ..., \tau_m \in (sorts(D) \cup Dom_{D,CS}) \wedge con : \tau_1 ... \tau_m \rightarrow \sigma \in CS \Rightarrow \sigma \in Dom_{D,CS}$ A* **comlpex constraint system** *is a constraint system that contains complex constraint domains.*

This possibly infinite set contains all types of variable domains that can be created by the constructors. In difference to the traditional approach, where all values have the same type, we cannot consider variable domains to be subsets of the constraint domain, but have to use a different universe for every type. Consequently, in our approach every object can only be instantiated with a restricted set of the possible values, namely those that have the correct type.

Definition 3. *Every* **complex constraint variable** *$x \in X$ in a complex constraint domain $Dom_{D,CS}$ which is constructed by any $cs \in CS$ with $cs : \sigma_1 \times ... \times \sigma_n \rightarrow \tau$ is associated to its domain type τ by the function $dom : X \rightarrow Dom_{D,CS}$.*

The actual domain of a complex constraint variable is a set, which is in most cases a Cartesian product. This set has to have the same structure as the variable's domain type. As the domain type is associated by a total mapping to every variable, it will always exist and be computable. To refer to the domain values themselves, which is performed by the constraints, the different components of the variable domain must be accessible. This can be done by the established concept of selectors. Selectors are functions that map complex structures to their internal components and are a built-in feature of most programming languages.

Proposition 2 *Given a set of constraint systems CS (as described in Prop. 1), that define the set $Dom_{D,CS}$ of complex constraint domains. A complex constraint system can be built upon these systems by the definition of a constraint system with the constructor $cs : \tau_1 ... \tau_n \rightarrow \sigma$, if $\tau_1, ..., \tau_n \in Dom_{D,CS}$ and the additional requirement that*

4. $\forall 1 \leq i \leq n : sel_i : \sigma \rightarrow \tau_i$ is defined

With these prerequisites, it is not necessary to actually handle the domains of complex variables. There is no need to construct the variable domains as

Cartesian products explicitly, because the internal sets can be accessed whenever values are required. Thus in difference to traditional constraint variables the domains of complex constraint variables are not represented in the respective application. The constraints, which access the variable domains, use selectors to access certain attributes of the complex variables and perform domain operations on the thus returned simple variable domains. The domains of the basic constraint variables are stored in a traditional way in the used constraint solver. This idea will become more clear, when I show, how constraints over complex variables can be implemented in the next Section. But first, I want to illustrate, how complex constraint domains can be constructed in the Algebraic Specification framework.

Example 3. The complex constraint domain of meetings (see Example 2) can be constructed from the constraint systems `DATE, TIME, DUR` and `LOC`.

```
MEETING = DATE + TIME + DUR + LOC +
 sorts: meeting
 opns: //constructor
   mkMeeting: date time dur loc -> meeting
  // selectors
  getDate: meeting -> date
  getTime: meeting -> time
  ...
   //constraints
  before: meeting meeting -> boolean
  shorter_than: meeting int -> boolean
  before_lunch: meeting -> boolean
```

The constraints (1),(3) and (4) can be implemented with some algorithm, that interprets the respective functions, the global constraint (2) cannot be implemented in this framework, because there is no datatype for lists or other accumulating structures available. We will show the specification of the constraints and their implementation later in this paper. The parameter systems (`DATE, TIME, DUR`) are constructed from `FD` (Example 1) and thus the basic type `INT`, I illustrate this in one case:

```
TIME = FD +
 sorts: time
 opns: mkTime: fd_var fd_var -> time //constructor
  getHour: time -> fd_var //selectors
  getMin: time -> fd_var
  before_time: time time -> boolean //constraint
```

The constraint system `LOC` is constructed from standard strings in a similar way as the system `FD`.

4 Constraint Solving in Complex Domains

Constraints over complex variables have, just as regular constraints, a declarative semantics, that restricts the allowed values for their variables. As the domains of complex variables are not explicitly constructed from the underlying sets, constraints over these variables can not be explicitly defined (as in 1. on page 3). Thus constraints over complex variables must be implemented with propagators (as described in 2. on page 3), that perform the intended domain restrictions. These propagators are to use the selectors of complex constraint variables to access the underlying data. Using this projection in every execution of the propagators, we omit the handling of complex constraint domains and can use the basic constraint systems as provided in many constraint solvers. Regular objects or other complex structures can then use constraint variables as attributes and the constraints will refer to the appropriate sub-structures.

Such complex structures can be easily fitted in any Object-Oriented model and still do not restrict the capabilities of constraint modeling. As mentioned before, constraints over such complex structures use selectors to get a reference to the required sub-structure and can then make domain restrictions on these attributes. In Example 4, we show, how the `before`-constraint, mentioned in previous examples can be implemented this way. I used the language CHR [7] to implement the constraints, which is integrated in the SICStus Prolog system. However, any implementation language and host system could be used, if it provides a means to implement constraints by propagators.

Example 4. The `before/2` constraint enforces the meeting represented by its first argument to be scheduled, such that it will be finished before the meeting in its second argument. As Prolog does not provide selector functions, pattern-matching is used in the heads of the rules instead.

```
before((D1,T1,Dur1,_),(D2,T2,_,_))
   <=> ground([D1,D2]),D1=D2  % if same date
   | before_time(T1,Dur1,T2). % time must be earlier
before((D1,_,_,_),(D2,_,_,_))
   <=> ground([D1,D2])  % otherwise,
   | before_day(D1,D2).   % date must be earlier
```

This constraint is reduced (by simplification rules) to constraints over the internal structures representing the date and the time. The constraint `before_time(T1,D1,T2)` enforces, that `T2` is later than `D1` minutes after `T1`. It is implemented with SICStus' builtin-constraint `#<`:

```
before_time((H1,M1),D1,(H2,M2))
   <=> ((H1 * 60) + M1) + D1 #< ((H2 * 60) + M2).
```

The benefit of such a constraint compared to a simple rule that invokes the underlying constraint `#<` is that constraint reasoning on the higher level is supported. In the above example, there is no need to model all timepoints on a single scale, such as "minutes after the Millennium" for example. Instead the commonly

understood, more complex order of timepoints represented by date *and* daytime can be used. This will allow a more direct link between input, constraint reasoning and output and has additional software technological advantages such as maintainability.

5 Conclusion

Complex Constraint Domains are motivated by the fact, that the restrictive capabilities of today's constraint solvers often have negative impact on the general software properties and ergonomics of constraint-based systems. This drawback can be made up with the use of constraints over complex constraint variables. These variables are models of real-world entities as opposed to traditional variables which are models of a predefined flat data structure.

The paper proposes a means to use Complex Constraint Domains in the Software Process of constraint based applications. This allows for standard software modeling methods, such as OOA or OOD, to be applied without restricting the capabilities of Constraint Programming. The software systems produced this way are more likely to meet general software requirements such as reusability or maintainability while preserving the powerful problem solving techniques of CP. Furthermore, this allows for better software ergonomics since the input format can be designed in accordance to the user's knowledge instead of the abstract mathematical model that can be handled by the used constraint solver.

References

[1] Krzysztof R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1998.

[2] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*, 1996.

[3] Alain Colmerauer. Multiplication constraint in several approximation spaces. Invited talk at CP01/ ILCP01 conference, Paphos, Cyprus, November 2001. `www.lim.univ-mrs.fr/~colmer/Transparents /Paphos01/paphose.ps`.

[4] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language chip. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS88)*, 1988.

[5] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1*. Springer-Verlag Berlin Heidelberg, 1985.

[6] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2*. Springer-Verlag Berlin Heidelberg, 1990.

[7] Thom Frühwirth. Theory and practice of constraint handling rules. *Special Issue on Constraint Logic Programming (P. Stuckey and K. Marriot, Eds.), Journal of Logic Programming*, 37(1-3):95–138, 1998.

[8] Thom Frühwirth and Slim Abdenadher. *Constraint-Programmierung*. Springer, Berlin Heidelberg, 1997. in german.

[9] Ian Gent and Toby Walsh. CSPLib, a problem library for constraints, `http://www-users.cs.york.ac.uk/~tw/csplib/`.

[10] J.-F. Puget. A c++ implementation of clp. Technical report, ILOG, 1994. In Ilog Solver Collected papers.

[11] Georg Ringwelski. *Asynchrones Constraintlösen*. PhD thesis, Technical University Berlin, 2003. `edocs.tu-berlin.de/diss/2003/ringwelski_georg.htm`.

[12] Eugene C. Freuder. Sarah O'Connell, Barry O'Sullivan. Query generation for interactive constraint acquisition. In *Proceedings of the 4th International Conference on Recent Advances in Soft Computing (RASC-2002)*, pages 295–300, 2002.

[13] Christian Schulte and Mats Carlson. Finite domain constraint programming systems. Tutorial at CP02 conference, Ithaca, USA, September 2002. `www.it.kth.se/schulte/talks/FD`.

[14] Ian Sommerville. *Software Engineering, 6th Edition*. Addison-Wesley, 2000.