The Proper Treatment of Undefinedness in Constraint Languages

Alan M. Frisch[1] and Peter J. Stuckey[2] *

[1] Artificial Intelligence Group, Dept. of Computer Science, Univ. of York, UK.
frisch@cs.york.ac.uk

[2] National ICT Australia. Dept. of Computer Science and Software Engineering, Univ. of Melbourne, Australia. pjs@csse.unimelb.edu.au

Abstract. Any sufficiently complex finite-domain constraint modelling language has the ability to express undefined values, for example division by zero, or array index out of bounds. This paper gives the first systematic treatment of undefinedness for finite-domain constraint languages. We present three alternative semantics for undefinedness, and for each of the semantics show how to map models that contain undefined expressions into equivalent models that do not. The resulting models can be implemented using existing constraint solving technology.

1 Introduction

Finite-domain constraint modelling languages enable us to express complicated satisfaction and optimization problems succinctly in a data independent way. Undefinedness arises in any reasonable constraint modelling language because, for convenience, modellers wish to use *functional syntax* to express their problems and in particular they want to be able to use *partial functions*. The two most common partial functions used in constraint models are division, which is undefined if the denominator is zero, and array lookup, $a[i]$, which is undefined if the value of i is outside the range of index values of a. Other partial functions available in some constraint modelling systems are square root, which is undefined on negative values, and exponentiation, which is undefined if the exponent is negative.

A survey of some existing constraint languages and solvers shows a bewildering pattern of behaviour in response to undefined expressions. Fig. 1 shows the results of solving five problems in which undefinedness arises with five finite-domain solvers: ECLiPSe 6.0 #42 [1], SWI Prolog 5.6.64 [2], SICStus Prolog 4.0.2 [3], OPL 6.2 [4] and MiniZinc 1.0 [5, 6]. The three rightmost columns, which are explained later in the paper, show the correct answers according to the three

* Much of this research was conducted while Alan Frisch was a visitor at the Univ. of Melbourne. His visit was supported by the Royal Society and the Univ. Melbourne. For many useful discussions we thank the members of the Essence and Zinc research teams, especially Chris Jefferson and Kim Marriott. We thank David Mitchell for advise about complexity. NICTA is funded by the Australian Government as represented by the Dept. of Broadband, Communications and the Digital Economy and the Australian Research Council.

Problem	ECLiPSe	SWI	SICStus	OPL	MiniZinc	Relational	Kleene	Strict
(1) $y \in \{0,1\}$ $1/y = 2 \ \vee \ y < 1$	$y \mapsto 0$	$y \mapsto 0$	none	none	$y \mapsto 0$	$y \mapsto 0$	$y \mapsto 0$	none
(2) $y \in \{-1,0\}$ $1 = \sqrt{y} \ \vee \ y < 0$	none					$y \mapsto -1$	$y \mapsto -1$	none
(3) $y \in \{3,4\}$ $a[y] = 1 \ \vee \ y > 3$ where a is $[1,4,9]$ and indexed 1..3				$y \mapsto 4$	$y \mapsto 4$	$y \mapsto 4$	$y \mapsto 4$	none
(4) $y \in \{0,1,2\}$ $\mathtt{T} \vee 1/y = 1$	$y \mapsto 0$ $y \mapsto 1$	$y \mapsto 0$ $y \mapsto 1$ $y \mapsto 2$	$y \mapsto 1$ $y \mapsto 2$	$y \mapsto 1$ $y \mapsto 2$	$y \mapsto 0$ $y \mapsto 1$ $y \mapsto 2$	$y \mapsto 0$ $y \mapsto 1$ $y \mapsto 2$	$y \mapsto 0$ $y \mapsto 1$ $y \mapsto 2$	$y \mapsto 1$ $y \mapsto 2$
(5) $y \in \{0,1\}$ $\neg(1/y = 1)$	$y \mapsto 0$	$y \mapsto 0$	none	none	$y \mapsto 0$	$y \mapsto 0$	none	none

Fig. 1. Examples of how undefinedness is handled.

semantics we introduce. Note that T is notation for *true*, and empty cells indicate that either the solver does not provide a square root function or that it does not allow an array to be accessed with a decision variable using functional notation. All five example problems involve Boolean operators ($\neg$ or $\vee$) because it is on such constraints that differences arise between and among implementations, our intuitions, and the three semantics introduced in this paper.

The first thing to notice is the disagreement among the solvers. The only compatible pairs of solvers are SICStus and OPL, and MiniZinc and SWI. The second observation is that some of the solvers behave irregularly. Problems (1), (2) and (3) are analogous—they just involve different partial functions—yet ECLiPSe finds a solution to (1) but not (2) and OPL finds a solution to (3) but not (1).

The issue of undefinedness in constraint languages and solvers has been the attention of almost no systematic thought. Consequently, as these examples show, implementations treat undefinedness in a rather haphazard manner and users do not know what behaviour to expect when undefinedness arises.

This paper directly confronts the two fundamental questions about undefinedness in constraint languages: What is the intended meaning of a model containing partial functions and how can those models be implemented? We address these questions by considering a simple modelling language, $\mathcal{E}$, that has two partial functions: division and array lookup. We first present three alternative truth-conditional semantics for the language: the relational semantics, the Kleene semantics, and the strict semantics. Each is obtained by starting with a simple intuition and pushing it systematically through the language. Following the standard convention that "$f(a) = \perp$" means that f is a partial function that is undefined on a, all three semantics use the value $\perp$ to represent the result of division by zero and out-of-bounds array lookups. The semantics differ in how other operators, including logical connectives and quantifiers, behave when applied to expressions that denote $\perp$. On models in which undefinedness does not arise, the semantics agree with each other, with existing implementations, and with our intuitions.

After presenting the three semantics for $\mathcal{E}$ we show how each can be implemented. Existing constraint modelling languages are implemented by mapping a constraint with nested operations into an existentially quantified conjunction

of un-nested, or flat, constraints. For example, $b \vee (x/y \geq z)$ gets mapped to

$$\exists\, b', t.\, (t = x/y) \wedge (b' = t \geq z) \wedge (\mathtt{T} = b \vee b').$$

Solvers then use libraries that provide procedures for propagating each of the flat constraints. Two difficulties confront attempts to use this approach when expressions can denote $\bot$. Firstly, existing propagation procedures do not handle $\bot$. For example, the propagator that handles $t = x/y$ can bind an integer value to t when the values of x and y are known and y is non-zero, but cannot bind $\bot$ to t if y is known to be zero. Secondly, the transformations that flatten nested constraints are equivalence preserving in classical logic, but some are not equivalence preserving in a non-classical semantics that uses $\bot$. For example, rewriting $\mathtt{T} \vee exp$ to $\mathtt{T}$ in not equivalence preserving for the strict semantics.

This paper employs a novel approach for implementing the three semantics for $\mathcal{E}$. Rather than transform constraints in $\mathcal{E}$ to flattened constraints, we transform the $\mathcal{E}$-model to another one that has the same solutions but in which undefinedness cannot arise. A different transformation is used for each of the semantics. Since undefinedness cannot arise in the resulting $\mathcal{E}$ models, they can be implemented using the well-understood techniques that are standardly used in the field.

2 A Simple Constraint Language

We use a simplified form of Essence [7], called $\mathcal{E}$, as our language for modelling decision problems, not just problem instances. Every model in $\mathcal{E}$ has exactly three statements, signalled by the keywords `given`, `find` and `such that`. As an example consider the following model of the graph colouring problem.

`given`	k:`int`, n:`int`, $Edge$: `array`$[1..n, 1..n]$ `of bool`
`find`	$Colour$: `array`$[1..n]$ `of int`$(1..k)$
`such that`	$\forall v{:}1..n-1.\ \forall v'{:}v..n.\ Edge[v, v'] \ \rightarrow\ Colour[v] \neq Colour[v']$

The `given` statement specifies three parameters: k, the number of colours; n, the number of vertices in the graph to be coloured; and $Edge$, an incidence matrix specifying the graph to be coloured. The integers $1..n$ represent the vertices of the graph. The `find` statement says that the goal of the problem is to find $Colour$, an array that has an integer $1..k$ for each vertex. Finally the `such that` statement requires that a solution must satisfy the constraint that for any two nodes, if there is an edge between them then they must have different colours.

The language has three main syntactic categories: statements, expressions and domains. Each expression of the language has a unique type that can be determined independently of where the expression appears. Where τ is a type we write $exp{:}\tau$ to denote an arbitrary expression of type τ. The types of the language are `int`, `bool`, and `array` $[IR]$ `of` τ, where τ is any type and IR is an integer range specifying the index values of the array. Throughout the language, an integer range, always denoted IR, is of the form exp_1:`int`$..exp_2$:`int` and never contains a decision variable. Notice that the array constructor can be nested; for example `array`$[1..10]$ `of array` $[0..5]$ `of int` is a type. We often abbreviate "`array` $[l_1..u_1]$ `of` $\cdots$ `of array` $[l_n..u_n]$" as "`array` $[l_1..l_n, \ldots, l_n..u_n]$."

Domains are used to associate a set of values with a parameter or decision variable. A domain is either (1) `bool`, (2) `int`, (3) of the form `int` (IR) or (4) of the form `array` $[IR]$ of Dom, where Dom is a domain. A domain is finite if it is constructed without using case (2) of the definition. The non-terminal $FDom$ is used for finite domains.

The syntax of the three statements is as follows (where $n \geq 0$):

`given` $NewId_1$:$Dom_1, \ldots, NewId_n$:Dom_n
  where Dom_i can contain an occurrence of $NewId_j$ only if $i \leq j$.
`find` $NewId_1$:$FDom_1, \ldots, NewId_n$:$FDom_n$
`such that` exp_1:`bool`, $\ldots, exp_n$:`bool`

Finally, let's consider the syntax of expressions, starting with the atomic expressions. The integer constants are written in the usual way and the constants of type `bool` are `T` and `F`. Each identifier that has been declared as a parameter or decision variable is an expression whose type is determined by the domain given in the declaration. A quantified variables can appear within the scope of its quantifier. As will be seen, quantified variables are always of type `int`.

The following are non-atomic expressions of type `int`:

- exp_1:`int` *intop* exp_2:`int`, where *intop* is one of $+$, $-$, $*$, or $/$,
- $-exp_1$:`int`,
- `boolToInt`(exp:`bool`), and
- $\sum NewId{:}IR.\ exp$:`int`

The symbol "/" is for integer division. An example of an integer expression using these constructs is $\sum i{:}0..n{-}1.\ \texttt{boolToInt}(a[i] = 0)$, which counts up the number of 0 entries in a .

The following are non-atomic expressions of type `bool`:

- exp_1:`bool` *boolop* exp_2:`bool`, where *boolop* is one of $\wedge$, $\vee$, $\rightarrow$ or $\leftrightarrow$.
- $\neg exp$:`bool`.
- exp_1:`int` *compop* exp_2:`int`, where *compop* is one of $=$, $\neq$, $\leq$ or $<$.
- $\mathcal{Q}\, NewId{:}IR.\ exp$:`bool`, where $\mathcal{Q}$ is a logical quantifier, $\exists$ or $\forall$.

Finally, the following expression is of type τ:

- $AR[exp$:`int`$]$, where AR is of type "`array` $[IR]$ `of` τ", for some τ. Notice that exp:`int` may contain free variables.

We often abbreviate "$AR[i_1]\cdots[i_n]$" as "$AR[i_1, \ldots, i_n]$".

For simplicity we assume that each identifier $NewId$ occurring in $\forall NewId{:}IR.\, exp$, $\exists NewId{:}IR.\, exp$ or $\sum NewId{:}IR.\, exp$ is a new identifier that appears nowhere else in the model except in exp.

3 The Semantics of $\mathcal{E}$

This section presents three alternative semantic accounts of $\mathcal{E}$. In each undefinedness arises in only two ways: dividing by zero and indexing into an array with a value that is out of bounds. The three accounts differ only in how they determine whether an expression is undefined if it contains an undefined subexpression. For models that are *safe*—those in which division by zero and out-of-bounds indices do not arise—the three semantics agree with each other and, we believe, with the intuitions of constraint modellers and the behaviour of constraint solvers. For safe models, solvers do not exhibit a haphazard pattern of behaviour.

This section first presents the part of the semantics that the three have in common. Then three subsections describe the distinctive parts of the three semantics.

The purpose of these semantics is to identify the solutions of an instance of an $\mathcal{E}$ model—that is, what assignments to decision variables satisfy what instances. To be clear, our focus is defining the truth conditions of the language, not on defining the behaviour of a decision procedure for satisfiability or any other program.

As the semantics defines solutions of instances, we start by defining the instances of a model: a pair $\langle M, I\rangle$ is a *problem instance* if M is an $\mathcal{E}$ model and I is an *instantiation* for M. An instantiation for M maps each parameter of M to a value that is appropriate as determined by the `given` statement of the model. If the `given` statement of M is "`given` $NewId_1$:$Dom_1, \ldots, NewId_n$:Dom_n", then an instantiation I of M maps each parameter $NewId_i$ to a member of the set denoted by Dom_i. The denotation of Dom_i, written $[\![Dom_i]\!]^I$, must be taken relative to I since Dom_i may itself contain parameters; for example in "`array`$[a * b..c * b]$ `of int`" the symbols a, b and c may be parameters. The following rules define the semantics of domains.

- $[\![exp_l..exp_u]\!]^I = \bot$ if $[\![exp_l]\!]^I = \bot$ or $[\![exp_u]\!]^I = \bot$ (1)
  $= \{i \in \mathbb{Z} \mid [\![exp_l]\!]^I \leq i \leq [\![exp_u]\!]^I\}$ otherwise
- $[\![\texttt{int}(IR)]\!]^I = \emptyset$ if $[\![IR]\!]^I = \bot$
  $= [\![IR]\!]^I$ otherwise
- $[\![\texttt{bool}]\!]^I = \{\texttt{T}, \texttt{F}\}$
- $[\![\texttt{int}]\!]^I = \mathbb{Z}$ (That is, the set of all integers.)
- $[\![\texttt{array}\ [IR]\ \texttt{of}\ DOM]\!]^I = \emptyset$ if $[\![IR]\!]^I = \bot$ or $[\![IR]\!]^I = \emptyset$
  $= [\![IR]\!]^I \longrightarrow [\![DOM]\!]^I$ otherwise.

Notice that an array denotes a total function over the set of index values that are within bounds. This set may be empty. Also notice that some models have no instantiations because a parameter may have a domain denoting the empty set.

The semantics must dictate whether an instance $\langle M, I\rangle$ of a model is satisfied by an assignment A to the decision variables of M. An assignment must map each decision variable to an appropriate value. If the `find` statement of M is "`find` $NewId_1$:$FDom_1, \ldots, NewId_n$:$FDom_n$", then an assignment A for $\langle M, I\rangle$ maps each decision variable $NewId_i$ to a member of $[\![FDom_i]\!]^I$. Notice that $[\![FDom_i]\!]^I$ may be the empty set, in which case the instance has no assignments and hence no solutions.

Finally, quantified variables are handled in the same way as in first-order logic—that is, denotations are taken relative to an assignment g that maps each quantified variable to an appropriate value.

We write $[\![M]\!]^{I,A,g}$ to mean the denotation of instance $\langle M, I\rangle$ with respect to A and g. As the denotation function is defined compositionally, we extend the notation and write $[\![exp]\!]^{I,A,g}$ where exp is any expression of $\mathcal{E}$.

Our primary intuition regarding undefinedness is that an assignment A is a solution to an instance if the constraints of the instance all denote `T` with respect

to the assignment, and this is the case even if the same constraints denote undefined with respect to other assignments. Thus we say that an instance I of model M is satisfied by assignment A if $[\![c]\!]^{I,A} = \mathtt{T}$ for every constraint c in M. This intuition would be violated by a solver that aborts if one assignment generates an error condition such as division by zero even though other assignments are solutions.

It remains to define the semantics of the expressions of $\mathcal{E}$. This section defines the semantics for expressions where they agree for all three semantics.

Let us start with the atomic expressions. In all assignments, "$\mathtt{T}$", "$\mathtt{F}$", "1", "2", "3", etc. denote $\mathtt{T}$, $\mathtt{F}$, 1, 2, 3, etc. For other atomic expressions we have:

- $[\![\alpha]\!]^{I,A,g} = I(\alpha)$ if α is a parameter
  $= A(\alpha)$ if α is a decision variable
  $= g(\alpha)$ if α is a quantified variable.

Now consider the operators that are used to build up integer expressions. For every binary integer operator *intop* we have

- $[\![exp_1 \ intop \ exp_2]\!]^{I,A,g} = \bot$ if $[\![exp_1]\!]^{I,A,g} = \bot$ or $[\![exp_2]\!]^{I,A,g} = \bot$
  $= [\![exp_1]\!]^{I,A,g} \ [\![intop]\!]^{I,A,g} \ [\![exp_2]\!]^{I,A,g}$ otherwise

where $[\![intop]\!]^{I,A,g}$ is the obvious operation. For division $[\![exp_1/exp_2]\!]^{I,A,g} = \bot$ if $[\![exp_2]\!]^{I,A,g} = 0$. Unary operators are handled in a similar manner.

Now consider summation expressions. If g is a variable assignment, then $\sigma[x \mapsto d]$ is the assignment that is identical to σ with the possible exception that it maps x to d.

- $[\![\sum x{:}IR.\ exp]\!]^{I,A,g} =$ if $[\![IR]\!]^{I,A,g} = \emptyset$ then 0
  else if $[\![IR]\!]^{I,A,g} = \bot$ then $\bot$
  else if $[\![exp]\!]^{I,A,g[x \mapsto d]} = \bot$ for some $d \in [\![IR]\!]^{I,A,g}$ then $\bot$
  else the sum of $[\![exp]\!]^{I,A,g[x \mapsto d]}$ for all $d \in [\![IR]\!]^{I,A,g}$

Notice that $\sum x{:}1..-1.\ x/0$ denotes 0 with respect to any assignments since $1..-1$ denotes the empty set.

The integer range associated with the summation quantifier and, indeed, all quantifiers, may contain free occurrences of quantified variables. So semantic rule (1) must be generalised to take assignments to quantified variables.

- $[\![exp_l..exp_u]\!]^{I,A,g} = \bot$ if $[\![exp_l]\!]^{I,A,g} = \bot$ or $[\![exp_u]\!]^{I,A,g} = \bot$
  $= \{i \in \mathbb{Z} \mid [\![exp_l]\!]^{I,A,g} \leq i \leq [\![exp_u]\!]^{I,A,g}\}$ otherwise

3.1 Semantics 1: A Three-Valued Kleene Semantics

This semantics follows the approach used by Frisch et. al. [8] in giving a semantics to Essence. Three truth values are used — $\mathtt{T}$, $\mathtt{F}$ and $\bot$ — where the intuition is that $\bot$ indicates a lack of information. Thus, $\mathtt{T} \vee \bot$ is $\mathtt{T}$ because it is $\mathtt{T}$ regardless of whether the "unknown" value $\bot$ is $\mathtt{T}$ or $\mathtt{F}$. Similarly, $\mathtt{T} \wedge \bot$ is $\bot$ because it could be $\mathtt{T}$ or $\mathtt{F}$ depending on the "unknown" value of the second argument. This results in the Boolean connectives of the three-valued propositional logic of Kleene [9, §64].

$\wedge$	$\mathtt{T}$	$\mathtt{F}$	$\bot$
$\mathtt{T}$	$\mathtt{T}$	$\mathtt{F}$	$\bot$
$\mathtt{F}$	$\mathtt{F}$	$\mathtt{F}$	$\mathtt{F}$
$\bot$	$\bot$	$\mathtt{F}$	$\bot$

$\vee$	$\mathtt{T}$	$\mathtt{F}$	$\bot$
$\mathtt{T}$	$\mathtt{T}$	$\mathtt{T}$	$\mathtt{T}$
$\mathtt{F}$	$\mathtt{T}$	$\mathtt{F}$	$\bot$
$\bot$	$\mathtt{T}$	$\bot$	$\bot$

$\rightarrow$	$\mathtt{T}$	$\mathtt{F}$	$\bot$
$\mathtt{T}$	$\mathtt{T}$	$\mathtt{F}$	$\bot$
$\mathtt{F}$	$\mathtt{T}$	$\mathtt{T}$	$\mathtt{T}$
$\bot$	$\mathtt{T}$	$\bot$	$\bot$

$\leftrightarrow$	$\mathtt{T}$	$\mathtt{F}$	$\bot$
$\mathtt{T}$	$\mathtt{T}$	$\mathtt{F}$	$\bot$
$\mathtt{F}$	$\mathtt{F}$	$\mathtt{T}$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\neg$	
$\mathtt{T}$	$\mathtt{F}$
$\mathtt{F}$	$\mathtt{T}$
$\bot$	$\bot$

`boolToInt`	
$\mathtt{T}$	1
$\mathtt{F}$	0
$\bot$	$\bot$

Existential quantification should behave like disjunction, which yields:

- $[\![\exists x{:}IR.\ exp{:}\texttt{bool}]\!]^{I,A,g}$
  $= \texttt{T}$ if $[\![IR]\!]^{I,A,g} \neq \bot$ and $[\![exp{:}\texttt{bool}]\!]^{I,A,g[x\mapsto d]} = \texttt{T}$ for some $d \in [\![IR]\!]^{I,A,g}$
  $= \texttt{F}$ if $[\![IR]\!]^{I,A,g} \neq \bot$ and $[\![exp{:}\texttt{bool}]\!]^{I,A,g[x\mapsto d]} = \texttt{F}$ for all $d \in [\![IR]\!]^{I,A,g}$
  $= \bot$ otherwise

The rule for universal quantification is obtained from this by interchanging "T" and "F." Notice that $\exists x{:}1..-1.\ 1/0 = 7$ denotes F with respect to any assignments since $1.. - 1$ denotes the empty set.

For integer comparison and array lookup we have:

- $[\![exp_1{:}\texttt{int}\ compop\ exp_2{:}\texttt{int}]\!]^{I,A,g} = \bot$ if $[\![exp_1]\!]^{I,A,g} = \bot$ or $[\![exp_2]\!]^{I,A,g} = \bot$
  $= [\![exp_1]\!]^{I,A,g}\ [\![compop]\!]^{I,A,g}\ [\![exp_2]\!]^{I,A,g}$ otherwise
- $[\![AR[exp{:}\texttt{int}]]\!]^{I,A,g} =$ if $[\![AR]\!]^{I,A,g} = \bot$ or $[\![exp{:}\texttt{int}]\!]^{I,A,g} = \bot$ then $\bot$
  else if the function $[\![AR]\!]^{I,A,g}$ is not defined on $[\![exp{:}\texttt{int}]\!]^{I,A,g}$ then $\bot$
  else $[\![AR]\!]^{I,A,g}([\![exp{:}\texttt{int}]\!]^{I,A,g})$

Again $[\![compop]\!]^{I,A,g}$ is the obvious operation.

3.2 Semantics 2: A Three-Valued Strict Semantics

The second semantic account of $\mathcal{E}$ is strict in that any compound expression is undefined whenever one of its sub-expressions is undefined. It is straightforward to specify the semantics based on this principle.

Here is the rule for the existential quantification, the remainder are similar:

- $[\![\exists x{:}IR.\ exp]\!]^{I,A,g} =$ if $[\![IR]\!]^{I,A,g} = \bot$ or $[\![exp]\!]^{I,A,g[x\mapsto d]} = \bot$ for some $d \in [\![IR]\!]^{I,A,g}$ then $\bot$
  else if $[\![exp]\!]^{I,A,g[x\mapsto d]} = \texttt{T}$ for some $d \in [\![IR]\!]^{I,A,g}$ then T
  else F

As in the Kleene semantics, a consequence of the rule for existential quantification is that $\exists x{:}1.. - 1.\ 1/0 = 7$ denotes F with respect to any assignment. Finally, the semantic rules for the comparison operators, the `boolToInt` operator and for indexing into arrays are the same as for the Kleene semantics.

3.3 Semantics 3: A Two-Valued Relational Semantics

The third semantic account for $\mathcal{E}$ is based on the observation that undefinedness results from the application of partial functions and the view that functional notation is a shorthand for relational notation. So, instead of thinking of division as a function, one could think of it as a relation, $\texttt{div}(x, y, z)$, which holds if and only if the result of dividing x by y is z (equivalently $y \times z = x \wedge y \neq 0$). Hence $\texttt{div}(5, 0, 3)$ denotes F not $\bot$.

In this semantics there can be undefined integer expressions, but all Boolean expressions are either T or F. Thus the Boolean operators $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$ as well as `boolToInt` have their usual classical interpretation. The rules for the logical quantifiers are:

- $[\![\exists x{:}IR.\ exp{:}\texttt{bool}]\!]^{I,A,g}$
  $= \texttt{T}$ if $[\![IR]\!]^{I,A,g} \neq \bot$ and $[\![exp{:}\texttt{bool}]\!]^{I,A,g[x\mapsto d]} = \texttt{T}$ for some $d \in [\![IR]\!]^{I,A,g}$
  $= \texttt{F}$ otherwise

The rule for indexing into an array AR of type "array[IR] of bool" is:

- $[\![AR[exp{:}\texttt{int}]]\!]^{I,A,g}$ = if $[\![AR]\!]^{I,A,g} = \bot$ or $[\![exp{:}\texttt{int}]\!]^{I,A,g} = \bot$ then F
  else if the function $[\![AR]\!]^{I,A,g}$ is not defined on$[\![exp{:}\texttt{int}]\!]^{I,A,g}$ then F
  else $[\![AR]\!]^{I,A,g}([\![exp{:}\texttt{int}]\!]^{I,A,g})$

If AR is an array of any other type, then the rule is the same as for the other two semantics.

3.4 Comparison of the Semantics

Here we briefly state some of the properties of and relationships among the three semantics. Space limitations preclude the presentation of examples or proofs. We invite the reader to revisit Fig. 1 and confirm that the three semantics produce the results shown in the last three columns.

Theorem 1. *Let e be any expression, I be any instantiation, A be any assignment and g be any variable assignment. Let s, k and r be the value of $[\![e]\!]^{I,A,g}$ in the strict, Kleene and relational semantics, respectively. If $s \neq \bot$ then $s = k$. If $k \neq \bot$ then $k = r$.* □

Theorem 2. *Let M be any $\mathcal{E}$ model. If in the strict semantics I is an instance of M and A is a solution to $\langle M, I\rangle$ then in the Kleene semantics I is an instance of M and A is a solution to $\langle M, I\rangle$. If in the Kleene semantics I is an instance of M and A is a solution to $\langle M, I\rangle$ then in the relational semantics I is an instance of M and A is a solution to $\langle M, I\rangle$.* □

In both the Kleene and relational semantics, a decision variable is the same as a prenexed existential quantifier. This is not the case in the strict semantics.

If e_1 and e_2 are integer expressions, then in both the Kleene and strict semantics $e_1 \neq e_2$ and $\neg(e_1 = e_2)$ are logically equivalent. Similarly, $e_1 < e_2$ and $\neg(e_2 \leq e_1)$ are logically equivalent in these two semantics. However, neither logical equivalence holds in general in the relational semantics.

In the Kleene and relational semantics, for every Boolean expression e, F and $\texttt{F} \wedge e$ are logically equivalent and T and $\texttt{T} \vee e$ are logically equivalent. Neither of these equivalences holds in general in the strict semantics.

In the Kleene and strict semantics, for every integer range IR and every Boolean expression ϕ, $\neg\forall x{:}IR.\ \phi$ and $\exists x{:}IR.\ \neg\phi$ are logically equivalent and $\forall x{:}IR.\ \neg\phi$ and $\neg\exists x{:}IR.\ \phi$ are logically equivalent. Neither equivalence generally holds in the relational semantics. For example, $\forall x{:}1/0..1/0.\ \neg(1 = 1)$ denotes F in all assignments but $\neg\exists x{:}1/0..1/0.\ 1 = 1$ denotes T in all assignments.

4 Transforming Constraints in $\mathcal{E}$

This section shows how, for each of the three semantics, a model can be transformed into one that has the same instances and solutions but whose constraints are *safe*. Space limitations require us to make the simplifying assumption that all expressions in `given` and `find` statements and in integer ranges associated with quantifiers are safe. More rigorously we say that an occurrence of an expression e is unsafe if $[\![e]\!]^{I,A,g} = \bot$ for some I, A and g. Furthermore an occurrence of an expression is unsafe if it contains any unsafe occurrence. Otherwise, an occurrence is said to be safe.

Let us first introduce the idea behind the transformations. Here we write $\phi[e]$ to denote an expression containing an occurrence of e. A subsequent reference to $\phi[e']$ denotes the same expression but with e replaced by e'. As an example to illustrate the transformations, consider transforming an atomic Boolean expression $A[e'/e]$ that is unsafe because e could denote 0. The expression $A[e'/e]$ may occur within a complex constraint.

In the relational semantics the basic idea is to transform $A[e'/e]$ to $\exists a'{:}\mathtt{nz}.\, a' = e \wedge A[e'/a']$. Here `nz` is the domain of all non-zero integers, so the resulting expression is safe. Notice that the resulting expression is false if e denotes 0.

In the strict semantics the basic idea is to transform $A[e'/e]$ in the same way as the relational semantics but also to add to the `such that` statement the constraint $e \neq 0$. This is a simplification because e may contain free variables.

The transformation for the Kleene semantics depends on the polarity of the occurrence of $A[e'/e]$. If it occurs in a positive context, then it is transformed in the same way as the relational semantics. However, if $A[e'/e]$ occurs in a negative context then it is transformed to $(\exists a'{:}\mathtt{nz}.\, a' = e \wedge A[e'/a']) \vee e = 0$. This expression is true if e denotes 0, which has the same effect as making the expression false in a positive context.

These basic transformations are logically correct except for the case `boolToInt` in the Kleene semantics, which is difficult to handle and requires special treatment. Unfortunately, these basic transformations add existential variables to the interior of a constraint and do not propagate efficiently. The actual transformations avoid these problems but, consequently, are much more complicated.

Now let's proceed to consider the actual transformations. In performing transformations to render a model safe, we need to pass through a language called $\mathcal{E}^+$, which is the same as $\mathcal{E}$ but with four additional features, each of which has the same denotation in each of the three semantics.

- A new kind of IR, called `nz`, which denotes the set of all non-zero integers.
- An additional *boolop*, $\Leftrightarrow$, where $x \Leftrightarrow y$ denotes `T` if x and y take the same value (including $\perp$) and `F` otherwise.
- The operator $\mathtt{boolToInt}_0$, which denotes the function that maps `T` to 1, `F` to 0, and $\perp$ to 0.
- A domain can contain "`array` $[l..u]$" where l and u can contain integer expressions of the form $\mathtt{MIN}\, x{:}IR.exp$ and $Qmax\, x{:}IR.exp$, respectively. If the IR associated with either form of expression denotes $\emptyset$ then the "`array` $[l..u]$" in which it occurs denotes an array with an empty set of indices. Otherwise, $[\![\mathtt{MIN}\, x{:}IR.exp]\!]^{I,A,g}$ and $[\![\mathtt{MIN}\, x{:}IR.exp]\!]^{I,A,g}$) are the minimum and maximum values of $\{[\![exp]\!]^{I,A,g[x\mapsto d]} | d \in [\![IR]\!]^{I,A,g}\}$.

Theorem 3. *The set of safe $\mathcal{E}^+$ models are the same in all three semantics. If M is a safe model, then its instantiations are the same in all three semantics and every instance of M has the same solutions in all three semantics.* □

TransformExtract(M)

While there is an occurrence A of a possibly unsafe Boolean expression in a constraint of M such that every occurrence of a Boolean expression within A is provably safe and the constraint does not contain an occurrence of "$\Leftrightarrow$" do:

Let b be an identifier that does not occur in M.
If no free quantified variables occur in A then
(a) Replace occurrence A in M with b.
(b) Add b:`bool` to the decision variables of M.
(c) Add to M the constraint $b \Leftrightarrow A$.

Otherwise
(d) Let $y_1, \ldots, y_m$ be, in order, the variables attached to the quantifiers whose scope includes A.
Let $y_{i_1}, \ldots, y_{i_n}$ be, in order, those variables in $y_1, \ldots, y_m$ that occur in A.
Let $l_i..u_i$ be the integer range expression over which each y_i is quantified.
Let MIN_j be the expression `MIN` $y_1{:}l_1..u_1. \cdots$ `MIN` $y_{j-1}{:}l_{j-1}..u_{j-1}. l_j$.
Let MAX_j be the expression `MAX` $y_1{:}l_1..u_1. \cdots$ `MAX` $y_{j-1}{:}l_{j-1}..u_{j-1}. u_j$.
(e) Replace occurrence A in M with $b[y_{i_1}, \ldots, y_{i_n}]$.
(f) Add "b:`array`$[MIN_{i_1}..MAX_{i_1}, \ldots, MIN_{i_n}..MAX_{i_n}]$`of bool`" to the decision variables of M.
(g) Add to M the constraint $\forall\, y_1{:}l_1..u_1. \cdots \forall y_n{:}l_n..u_n.\ b[y_{i_1}, \ldots, y_{i_n}] \Leftrightarrow A$

Fig. 2. TransformExtract.

By reductions from Diophantine problems it is straightforward to show that neither the safe nor the unsafe expressions of $\mathcal{E}$ are recursively enumerable. Therefore, for the transformations we assume the existence of a procedure that can determine that some expressions are safe, though it can not detect all safe expressions. We place three requirements on such a procedure: (1) if the procedure says that an expression is safe, then it is safe; (2) the procedure identifies as safe every expression of the form exp/i, where i is an integer variable with domain `nz`; and (3) the procedure identifies as safe every expression of the form and $AR[i]$, where i is an integer variable and the domain of i and the index range of AR are defined with syntactically identical expressions. If this procedure identifies an expression as safe, then we say that the expression is *provably safe*, otherwise we say that it is *possibly unsafe*.

Our transforms work by eliminating all possibly unsafe expressions and are correct even when the eliminated expression happens to be safe. However, a more accurate estimate of safeness results in the transformations making fewer changes to the model, thus producing a simpler model.

Example 1. Consider the model in $\mathcal{E}$ of the form:

`given` m: `array`[1..10] `of int`(1..5)
`find` x:`int`(1..20), y:`int`(−3..3)
`such that` $(\forall j{:}1..4.\ \sum i{:}j..9.\ \texttt{boolToInt}(m[i] \geq m[x]) \leq j/(y^2-5))$

then the expression $m[i]$ is provably safe, and we may assume expressions $m[x]$ and $j/(y^2-5)$ are possibly unsafe, although a more sophisticated analysis could determine that $y^2 - 5$ cannot take the value 0. □

The transforms for all three semantics use the common sub-procedure TransformExtract, given in Fig. 2, to transform a model M.

Example 2. Consider applying TransformExtract to the following model:

$$\begin{array}{ll}\texttt{find} & x\texttt{:int}-1..10\\ \texttt{such that} & 1 \leq 1/\texttt{boolToInt}(7/x \leq 1)\end{array}$$

First A must be chosen to be $7/x \leq 1$. (The transform cannot choose A to be the entire constraint as this contains a possibly unsafe Boolean expression.) Steps (a), (b) and (c) are executed, resulting in

$$\begin{array}{ll}\texttt{find} & x\texttt{:int}-1..10, b_1\texttt{:bool}\\ \texttt{such that} & 1 \leq 1/\texttt{boolToInt}(b_1),\\ & b_1 \Leftrightarrow 7/x \leq 1\end{array}$$

Next A is chosen to be $1 \leq 1/\texttt{boolToInt}(b_1)$ and steps (a), (b) and (c) are executed, resulting in

$$\begin{array}{ll}\texttt{find} & x\texttt{:int}-1..10, b_1\texttt{:bool}, b_2\texttt{:bool}\\ \texttt{such that} & b_2,\\ & b_2 \Leftrightarrow 1 \leq 1/\texttt{boolToInt}(b_1),\\ & b_1 \Leftrightarrow 7/x \leq 1\end{array}$$

□

Let us consider the relationship between a model M and the model M' that results from applying TransformExtract to M. We say that an assignment α' is an extension of an assignment α if every variable assigned by α is also assigned by α' and the two assign the same values to the variables of α.

Theorem 4. *Let M be a model and M' be the result of applying TransformExtract to M. Let I be any instantiation for M (and hence for M'). Then in any of the three semantics, assignment α is a solution to $\langle M, I\rangle$ if and only if some extension of α is a solution to $\langle M', I\rangle$.* □

After performing TransformExtract a model consists of two disjoint sets of constraints: M', the original constraints modified by steps (a) and (e) of TransformExtract, and B, the set of constraints added by steps (c) and (g) of TransformExtract. Notice that every constraint in M' is provably safe and every constraint in B is possibly unsafe. The transformation needed to make B provably safe is different for each of the three semantics. We consider each in turn.

4.1 Transformations for the Relational Semantics

To obtain a safe model for the relational semantics, Transform2Rel, as shown in Fig. 3 is performed. The transformations introduce a new variable a' to take the place of *exp*, and a new Boolean b' to capture whether $a' = exp$. a' has a domain that forces the resulting expression to be provably safe. The complexity arises in capturing the cases where a' and *exp* differ in value. If $a' \neq exp$ then the Boolean b is forced to be `F`. The third conjunct is required since otherwise we could choose $a' \neq exp$ and make b `F` when indeed the expression will not lead to undefined. The third conjunct forces $a' = exp$ if this will not result in an undefined expression.

Theorem 5. *Let M be a model resulting from the application of TransformExtract and let M' be the result of applying Transform2Rel to M. Let I be any instantiation for M (and hence for M'). Then M' is safe and in the relational semantics $\langle M', I\rangle$ and $\langle M, I\rangle$ have the same solutions.* □

Transform2Rel(M)

(a) Perform TransformExtract(M).
(b) While M contains a possibly unsafe occurrence of $b \Leftrightarrow C$ perform Transform2Pos($M, b \Leftrightarrow C$).

Transform2Pos($M, b \Leftrightarrow C$)

(c) If C contains an possibly unsafe expression of the form exp'/exp where exp is a provably safe expression then replace $b \Leftrightarrow C$ with

$$\exists\, a'{:}\mathtt{nz}.\ \exists\, b'{:}\mathtt{bool}.\ \ b' \Leftrightarrow (a' = exp) \wedge b \Leftrightarrow (b' \wedge C\{exp \mapsto a'\}) \wedge exp \neq 0 \rightarrow b'$$

(d) If C contains a possibly unsafe expression of the form $AR[exp]$, where AR is an expression of type `array` [l..u] of τ, and exp is a provably safe expression then replace $b \Leftrightarrow C$ with

$$\exists\, a'{:}l..u.\ \exists\, b'{:}\mathtt{bool}.\ \ b' \Leftrightarrow (a' = exp) \wedge b \Leftrightarrow (b' \wedge C\{exp \mapsto a'\}) \wedge (l \leq exp \wedge exp \leq u) \rightarrow b'$$

Fig. 3. Transform2Rel.

4.2 Transformations for the Strict Semantics

The strict semantics is the simplest to implement. The full transformation is given in Fig. 4.

Theorem 6. *Let M be a model resulting from the application of TransformExtract and let M' be the result of applying Transform2Strict to M. Let I be any instantiation for M (and hence for M'). Then M' is safe and in the strict semantics $\langle M', I\rangle$ and $\langle M, I\rangle$ have the same solutions.* □

4.3 Transformations for the Kleene Semantics

The Kleene semantics is the most difficult to make safe. This is the only semantics where Boolean expressions can really take the value $\perp$ (in the strict semantics if this occurs then there can be no solution). In order to transform this correctly to an effectively two valued semantics that is supported by the underlying constraint solvers we need to take into account whether a Boolean expression occurs in a positive context, where undefined will be equivalent to `F` for satisfiability, or a negative context where undefined will be equivalent to `T` for satisfiability.

Transform2Strict(M)

(a) Perform TransformExtract(M).
(b) While M contains a possibly unsafe occurrence of $b \Leftrightarrow C$ do:
  (c) If C contains a possibly unsafe expression of the form exp'/exp, where exp is provably safe then replace $b \Leftrightarrow C$ with

$$\exists\, a'{:}\mathtt{nz}.\ a' = exp \wedge (b \Leftrightarrow C\{exp \mapsto a'\})$$

  (d) If C contains a possibly unsafe expression of the form $AR[exp]$, where AR is an expression of type `array` [l..u] of τ, then replace $b \Leftrightarrow C$ with

$$\exists\, a'{:}l..u\ .\ a' = exp \wedge (b \Leftrightarrow C\{exp \mapsto a'\})$$

Fig. 4. Transform2Strict.

Transform2Kleene(M)

(a) Replace every occurrence in M of an expression of the form $exp_1 \leftrightarrow exp_2$ with $(exp_1 \rightarrow exp_2) \wedge (exp_2 \rightarrow exp_1)$.
(b) Replace every occurrence in M of an expression of the form $\texttt{boolToInt}(exp)$ with $\texttt{boolToInt}_0(exp)/(\texttt{boolToInt}_0(exp) + \texttt{boolToInt}_0(\neg exp))$.
(c) Perform TransformExtract(M)
(d) While M contains a possibly unsafe occurrence of $b \Leftrightarrow C$ do:
  (e) if b occurs positively in M then perform Transform2Pos($M, b \Leftrightarrow C$)
  (f) otherwise perform Transform2Neg($M, b \Leftrightarrow C$).

Transform2Neg($M, b \Leftrightarrow C$)

(g) If C contains a possibly unsafe expression of the form exp'/exp where exp is provably safe then replace $b \Leftrightarrow C$ with

$$\begin{aligned} \exists a'{:}\texttt{nz}.\ \exists b'{:}\texttt{bool}.\ \exists b''{:}\texttt{bool}.\ \ & b' \Leftrightarrow (a' = exp) \wedge \\ & b'' \Leftrightarrow (exp \neq 0) \wedge \\ & b \Leftrightarrow (\neg b'' \vee (b' \wedge C\{exp \mapsto a'\})) \wedge \\ & b'' \rightarrow b' \end{aligned}$$

(h) If C contains a possibly unsafe expression of the form $AR[exp]$, where AR is an expression of type $\texttt{array}\ [\mathrm{l..u}]$ of τ and exp is provably safe, then replace $b \Leftrightarrow C$ with

$$\begin{aligned} \exists a'{:}l..u.\ \exists b'{:}\texttt{bool}.\ \exists b''{:}\texttt{bool}.\ \ & b' \Leftrightarrow (a' = exp) \wedge \\ & b'' \Leftrightarrow (l \leq exp \wedge exp \leq u) \\ & b \Leftrightarrow (\neg b'' \vee (b' \wedge C\{exp \mapsto a'\})) \wedge \\ & b'' \rightarrow b' \end{aligned}$$

Fig. 5. Transform2Kleene

The Transform2Kleene transformation, shown in Fig. 5, converts an $\mathcal{E}$ model into a safe $\mathcal{E}^+$ model. Step (a) replaces each occurrence of $\leftrightarrow$ with two implications. The effect, as will be seen, is that every expression appears in either a positive or negative context, but not both. Step (b) replaces each occurrence of `boolToInt` with an equivalent expression containing two occurrences of $\texttt{boolToInt}_0$. This is done because the remainder of the transformation correctly deals with $\texttt{boolToInt}_o$ without any special provisions. Though each of these first two steps can make the model exponentially larger, a more sophisticated version of these steps could avoid this by introducing new Boolean variables. Step (c) performs TransformExtract, just as in the other two transforms. Finally Step (d) replaces all possibly unsafe expressions with ones that are provably safe. Positive occurrences are handled as in the relational transformation; negative occurrences employ a new transformation, Transform2Neg, also shown in Fig. 5.

Given an $\mathcal{E}^+$ expression which does not include or `boolToInt` or $\leftrightarrow$ we can define the context of each Boolean expression appearing in a model as follows.

- For `such that` $exp_1, \ldots, exp_n$ each of $exp_i, 1 \leq i \leq n$ appear positively.
- For $\texttt{boolToInt}_0(exp)$ then exp appears positively.
- If $\neg exp$ appears positively then exp appears negatively, and if $\neg exp$ appears negatively then exp appears positively.
- If $exp \vee exp'$ or $exp \wedge exp'$ appear in manner H (positively or negatively) then exp and exp' appear in manner H.
- If b appears in manner H (positively or negatively) and $b \Leftrightarrow exp$ occurs in M then exp appears in manner H.

Note that since the model results from TransformExtract the rules for expressions of the form $b \Leftrightarrow C$ are unambiguous since there is exactly one such expression for each introduced b.

Theorem 7. *Let M be a model resulting from the application of TransformExtract. Let M' be the result of applying Transform2Kleene to M. Then Let I be any instantiation for M (and hence for M'). Then M' is safe and in the Kleene semantics $\langle M', I\rangle$ and $\langle M, I\rangle$ have the same solutions.* □

5 Solving the Transformed Models

The transformations defined in the previous section create models that are safe; undefinedness cannot occur. we can now replace $\Leftrightarrow$ by $\leftrightarrow$ and $\texttt{boolToInt}_0$ by `boolToInt` since these operators are identical in two-valued logic. The models are still not directly executable in a constraint solver which takes a set of finite-domain variables and conjunction of constraints on these variables. In order to create such a final form we need to map the resulting model in $\mathcal{E}$ further, principally unrolling loops and flattening. Since the models are safe the existing mappings should respect the semantics of the model.

The reader may be concerned that the transforms introduce new variables with the infinite domain `nz`. This is unproblematic since it can be shown that if a search assigns values to all the finite-domain variables, then the value of each infinite-domain variable either becomes irrelevant to determining satisfiability or is fixed by propagation (even using simple propagators). As an example, consider the variable a' introduced by line (c) of Transform2Pos. If search fixes the value of exp to 0 and then propagation on $b' \Leftrightarrow a' = exp$ fixes b' to `F` and propagation on $b' \Leftrightarrow a' = exp$ can fix b to `F` without knowing the value of a'. On the other hand, if search fixes exp to a value other than 0, then propagation on $exp \neq 0 \rightarrow b'$ fixes b' to `T` and propagation on $b \Leftrightarrow (a' = exp)$ forces a' to be fixed to the same value as exp.

It can be shown that with only weak assumptions about propagators, none of the transformations weaken propagation. As an example, consider the model:

`find`	x:`int`$(1..6)$
`such that`	$\neg(12/x \geq 4)$

Assuming this model is implemented by the constraint $div(12, x, t) \wedge (b \leftrightarrow t \geq 4) \wedge \neg b$, then enforcing domain consistency results in the domains: $t : 2..3$, $x : 4..6$.

The model is safe so there is no need to transform it. However if we did transform it for the relational semantics then the result, after converting existential variables to decision variables, would be

`find`	x:`int`$(1..6)$, a':`nz`, b:`bool`, b':`bool`
`such that`	$b' \Leftrightarrow (a' = x) \wedge b \Leftrightarrow (b' \wedge (12/a' \geq 4)) \wedge (x \neq 0 \rightarrow b') \wedge \neg b$

Assuming this model is implemented by the constraint $(b \leftrightarrow b' \wedge b_2) \wedge (b' \leftrightarrow a' = x) \wedge (b_2 \leftrightarrow t \geq 4) \wedge div(12, a', t) \wedge (b_3 \rightarrow b') \wedge (b_3 \leftrightarrow x \neq 0) \wedge \neg b$ then enforcing domain consistency results in the domains: $b = \texttt{F}$, $b_3 = \texttt{T}$, $b' = \texttt{T}$, $b_2 = \texttt{F}$, $t : 2..3$, $a' : 4..6$, and $x : 4..6$.

6 Conclusion

As modelling languages become more expressive, it becomes more likely that a modeller creates models where undefinedness occurs. A clear understanding of how undefinedness is treated by a modelling language is vital to both the modeller and the system's implementer. For the modeller misunderstanding may result in modelling errors giving the wrong results (including incorrect optimal values) when undefinedness is silently translated to failure. For the systems builder a great deal of care must be taken to ensure that transformations and optimizations of the systems do not change the meaning of the model.

Fig. 1 shows that without a clear understanding of undefinedness implementers have struggled to implement a proper treatment of undefinedness. Our work shows how undefinedness can be properly treated in a simple constraint language and we believe that this approach can be extended easily to handle richer languages. Already our work has informed the development of MiniZinc and generated a bug report for ECLiPSe. We expect our work to result in bug reports in additional languages and to alter and inform the development of other languages such as Essence′. Finally, other subtle issues are likely to arise in the development of highly-expressive modelling languages; our work suggests that the use of semantics could be valuable in resolving those issues.

References

1. Apt, K.R., Wallace, M.: Constraint Logic Programming Using ECLiPSe. Cambridge University Press (2006)
2. SWI Prolog: http://www.swi-prolog.org/ (2009)
3. Intelligent Systems Laboratory: SICStus Prolog. Swedish Institute of Computer Science. (2009) http://www.sics.se/isl/sicstuswww/site/index.html.
4. Van Hentenrcyk, P.: The OPL Optimization Programming Language. MIT Press (1999)
5. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G.: Minizinc: Towards a standard CP modelling language. In: Proc. of the 13th Int. Conf. on Principles and Practice of Constraint Programming, Springer (2007) 529–543
6. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P.J., Garcia de la Banda, M., Wallace, M.: The design of the Zinc modelling language. Constraints **13** (2008) 229–267
7. Frisch, A.M., Harvey, W., Jefferson, C., Hernández, B.M., Miguel, I.: Essence : A constraint language for specifying combinatorial problems. Constraints **13** (2008) 268–306
8. Frisch, A.M., Grum, M., Jefferson, C., Martínez Hernández, B., Miguel, I.: The essence of Essence: A language for specifying combinatorial problems. In: Proc. of the 4th Int. Workshop on Modelling and Reformulating Constraint Satisfaction Problems. (2005) 73–88
9. Kleene, S.C.: Introduction to Metamathematics. Van Nostrand (1952)