How People Visually Represent Discrete Constraint Problems

Xu Zhu, Miguel A. Nacenta, Özgür Akgün, and Peter Nightingale

Abstract—Problems such as timetabling or personnel allocation can be modeled and solved using discrete constraint programming languages. However, while existing constraint solving software solves such problems quickly in many cases, these systems involve specialized languages that require significant time and effort to learn and apply. These languages are typically text-based and often difficult to interpret and understand quickly, especially for people without engineering or mathematics backgrounds. Visualization could provide an alternative way to model and understand such problems. Although many visual programming languages exist for procedural languages, visual encoding of problem specifications has not received much attention. Future problem visualization languages could represent problem elements and their constraints unambiguously, but without unnecessary cognitive burdens for those needing to translate their problem's mental representation into diagrams. As a first step towards such languages, we executed a study that catalogs how people represent constraint problems graphically. We studied three groups with different expertise: non-computer scientists, computer scientists and constraint programmers and analyzed their marks on paper (e.g., arrows), gestures (e.g., pointing) and the mappings to problem concepts (e.g., containers, sets). We provide foundations to guide future tool designs allowing people to effectively grasp, model and solve problems through visual representations.

Index Terms—Problem Visualization, Problem Modeling, Problem Solving, Constraint Programming, Visual Programming Languages

1 Introduction

People encounter constraint problems often in their daily lives. For example, one might have to create a schedule for a conference in which some events should take place before other events, avoid certain times, and several other constraints. One of the first steps in the process of solving such problems, as highlighted by the problem solving expert George Pólya, is to represent the problem [63, III].

The potential and importance of appropriate representations of problems is difficult to overstate. An effective description of the problem can be useful to communicate the problem to others or to ourselves at a later point in time. We also know that *how* the problem is represented might have a significant effect on a human's ability to solve it (e.g., [1, 87]). There are multiple examples of notations (representational systems) considered key in the advancement of areas of science (some graphical examples include Feynmann diagrams [34] and Penrose graphical notation [62]). Finally, if software exists that can help solve the problem, a representation of the problem becomes a key element of the interface.

One way of representing problems is through visual representation. Visuals can facilitate problem understanding (one could understand a problem faster and more precisely), communication (a common language of problem description can avoid misunderstanding between people) and in human-machine interaction (a sufficiently precise language would enable people to create problem specifications that can be interpreted and solved by a computer). Although there has been a large amount of research in InfoVis about how visual *data* representations are perceived, understood and interacted with, we know relatively little about how to build effective *problem* visualizations.

In this paper, we gain understanding of how to build problem visualizations by asking people to visually represent problems in their own way. Our explicit assumption is that understanding how people naturally represent problems will benefit the designs of languages for problem description. For example, a language designed with this knowledge in mind could be easier to learn and to translate problems into. Instead of addressing all types of problems, we start by looking only at discrete constraint problems.

We asked 30 participants with three different levels of formal programming expertise (non-computer scientists, computer scientists who are not constraint programmers and constraint programmers) to sketch visual representations of constraint problems using pen, paper, scissors and colour pens. We analyzed their representations and the videos of their processes. From these we generated a tree of visual elements and a tree of parts of problem language (problem concepts that participants visualized) that support a semiotic analysis.

Our analysis provides a first picture of how people with different levels of formal training in programming approach the task of describing problems. We measured the variability of mappings, catalogued regularities, and selected insights grouped around four main issues: diagrams and mathematics use; containers and symbols vs. textual labels; the problem of abstract representation through graphical means; and implicit information.

At this stage of the research we cannot make inferential quantitative claims that are generalizable to the full population (due to the methodology, the sample size, and the inability to estimate the statistical reliability of our prevalence and entropy measurements). Nevertheless, our findings can inform designers of notations and visual languages dealing with the representation of constraint problems and enable the design of visual notations, languages and interfaces that are easier to learn, faster to understand and are accessible to a wider set of the general population. Some of our findings may also have applicability beyond constraint problems (see Section 8.4).

2 Example Problem

To give the reader a more concrete idea of what we refer to as a "problem", and to show the type of problem associated to the tasks that our participant completed (problems modellable through constraint programming), we present here the *knapsack* problem:

Given a knapsack of capacity X and a set of objects, each with a specified volume and price, place items in the knapsack to maximize the total value without exceeding the capacity [47].

- *Xu Zhu is with the School of Computer Science, University of St Andrews, Jack Cole Building, North Haugh, St Andrews, KY16 9SX, United Kingdom.* mailto:xz32@st-andrews.ac.uk
- *Miguel A. Nacenta is with the School of Computer Science, University of St Andrews, Jack Cole Building, North Haugh, St Andrews, KY16 9SX, United Kingdom.* mailto:mans@st-andrews.ac.uk
- *Özgür Akgün is with the School of Computer Science, University of St Andrews, Jack Cole Building, North Haugh, St Andrews, KY16 9SX, United Kingdom.* mailto:ozgur.akgun@st-andrews.ac.uk
- *Peter Nightingale is with the School of Computer Science, University of St Andrews, Jack Cole Building, North Haugh, St Andrews, KY16 9SX, United Kingdom, and with the Department of Computer Science, University of York, Deramore Lane, Heslington, York, YO10 5GH, United Kingdom.* mailto:pwn1@st-andrews.ac.uk

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

This problem might seem contrived, but it is equivalent to problems people encounter in their professional and daily lives such as truck loading. An example model in Essence appears in csplib, problem 133 [33].

3 Research Goal, Scope and Questions

Our main goal is to provide information (e.g., a catalog of regularities, guidelines) that can help design notations and problem specification visual languages that are easily learnable, understandable and effective.

As the space of problems that people may encounter is large, we restrict ourselves to studying problems that can be easily described as *discrete constraint optimization problems*, as defined in [69, Chap. 3]. Admittedly, these are a subset of constraints problems (not addressed types include geometric and layout [4], graphs [30, 83], and continuous constraints). The reason for this is three-fold: a) a mature set of software tools (such as ECLiPSe Prolog [3], MiniZinc [57], Savile Row [58] and Minion [20]) exist that can efficiently find solutions; b) this problem family is particularly suitable as such problems are common and relevant in many areas of human activity (e.g., timetabling, resource and job allocation, and even common puzzles [33]); and, c) the aforementioned software solvers are inaccessible for non-experts.

In addition, because the way in which people represent problems is influenced by their experience and formal education, we decided to look at three cohorts of people representing a range of levels of familiarity with formal problem specification: non-computer scientists (Non-CS), computer scientists (CS), and constraint programmers (CP).

Based on this goal and scope, we aim to address these questions:

Q1: Which graphical elements do people choose to externalize problem constructs?

Q2: Which constructs do people choose to represent?

Q3: Which patterns appear in how people and different cohorts visually represent problems?

4 Background and Related Work

Our work connects several domains of knowledge and research. We discuss problem solving and modeling, visual notations, sketching and visual languages, as well as constraint programming and related techniques. We also introduce semiotic analysis as a technique.

4.1 Problem solving and modeling

Because problem solving is a ubiquitous human activity it has received ample attention from multiple disciplines, including psychology and neuroscience (e.g., [68]) and mathematics (e.g., [63]). Existing research recognizes problem representation as one of the key elements or stages for solving a problem, and scientists often propose notations as a way to advance their fields [34, 62]. Writing and sketching are often seen to be a natural extension of internal mental processes and help to augment human memory and processing capacity [75], which has been studied also for the visualization of data [82].

The role of representation has also been studied in educational contexts (e.g., [38]). Despite this, most externalized problem representation notations are designed ad-hoc for specific problems, and little attention has been paid to how people construct these problem representations. In contrast, there is a significant amount of work in understanding how people build models of working systems (e.g., physical or economic systems [21]) and, more recently, the role of models in understanding data through visualization [43].

4.2 Visual Notations, Sketching and Visual Languages

Circuit design and manufacturing has long been supported by the use of electrical and electronic diagrams. More recently, the growing complexity of software spurred the development of much work on supporting the design and understanding of programs, culminating with the most widely used software specification language, UML [71]. These languages typically specify architectures, structure of systems, and instances of user behaviour, but usually not problems. In methodologically related work (although with a different focus), Walny et al. and Cherubini et al. have studied how software engineers use sketching to support their thinking processes [14, 80, 81]. Our study shares with these the classification and categorization of sketches.

Simultaneously, many visual programming languages have been proposed [11, 29, 55], often as attempts to make programming accessible to broader audiences (e.g., [53, 67, 70]), or to manage the complexity of specifying systems that are highly interconnected [16, 48]. Visual programming languages are typically procedural rather than descriptive or declarative and are not free from their own limitations such as scalability [12] and clutter [56, 65]. Some classical drawing and simulation tools like Sketchpad and ThingLab also provide a graphical interface while including geometric or simple numerical constraints. [9, 10, 74] Theoretical aspects of the design, parsing and specification of visual programming languages are extensively discussed by Marriott and Meyer [46].

4.3 Constraint programming and related areas

A constraint problem is a problem that can be expressed through variables and a set of constraints. A constraint is a rule that expresses the allowable values of variables or of their relationships. Constraint Programming [69] and closely-related techniques such as Integer Linear Programming (ILP) [28] and Propositional Satisfiability (SAT) [8] are declarative methods for stating and solving this kind of discrete decision-making and optimization problem. Each technique has strengths and weaknesses in solving efficiency. Hybrid solvers such as SCIP [23] (hybridizing ILP and constraint programming) are increasingly common. Constraint programming is successfully applied in many high-impact areas such as timetabling, staff rostering, logistics, production planning and experiment design [64, 79].

The process of applying constraint programming to a problem can be crudely divided into two parts: *modeling* and *solving*. Once a problem is modeled into a suitable language, it can be automatically solved using a standard constraint solver. For complex real life problems, the modeling step presents a real difficulty: capturing a correct and efficient model is hard, even for experts. High-level modeling languages like Essence [2, 19] and Zinc [17] reduce the need for this expertise somewhat through abstract domain types like sets, functions, and relations.

There are several examples of visualization tools for the *solving* process, we describe a small selection here. Bauer et al. [6] presented an integrated development environment (IDE) for constraint programming. The IDE provides a visual debugger which displays the search tree that is explored by the constraint solver. The debugger is solver-independent, with minor modifications it can support any solver. However their system only focuses on visualizing the solving process and not modeling. Recently Goodwin et al. [25] described a user-centered design process for tools that visualize the solving process, building on earlier work by Shishmarev et al. [72]. From an Information Visualization perspective Goodwin et al. [25] looked at how different visualizations could be useful in the process of profiling constraint models. In addition, tools for layout constraints such as Auto Layout are also prevalent in IDEs. To the best of our knowledge, our paper is the first on the topic of visualization for the modeling phase of constraint programming.

4.4 Semiotic analysis

In this article we look at people's problem description ability using the basic concepts of early semiological analysis as initially proposed by Saussure [18], who defines symbol systems as mappings between signs (signifiers) and the signified. In his famous *Semiology of Graphics* [7] Bertin dissects, among other things, the mappings between elements on the page (marks) and data. More recently, Horn [31] has performed a semiotic analysis of the multimedia signals used in popular and business communication. Although we borrow from Bertin and Horn, we instead look at the relationship between signs in the page and gestures and elements of problem descriptions.

5 Methodology

We designed a controlled observation of people representing and trying to solve constraint type problems. We describe all the aspects of the empirical design although we will only briefly refer to the solving phases of the study, since this analysis does not fall within our remit.

5.1 Participants

We recruited 30 participants, 10 belonging to each of the three expertise groups, all from a local university. Non-CS participants (7 female, between 19 and 28 years in age), were non-computer scientists with little or no programming experience. Computer scientists (4 females, between 19 and 42 years in age) were students in a computer science degree with little or no experience in constraints programming but with experience of computer programming. Constraint programmers (CP) (1 female, between 21 and 64 years in age), were a mixture of students and staff who have either taken a constraints programming module, taught one or conduct research in that area. Participants received gift vouchers for their time. The three distinct groups were chosen because the way in which people represent problems is likely to be influenced by their experience and formal education, and they represent a range of levels of familiarity with formal problem specifications.

5.2 Procedure, Tasks and Problem Selection

Each participant provided written consent and was then assigned two problems. Problems were selected from a pool of constraint problems collected from csplib [33] as well as suggested by constraint programming experts. We selected six problems according to the following criteria: a) should not be too difficult to understand by a non-programming person; b) should contain elements that are familiar to most people; c) problems that are familiar to the general public are preferable; and d) should cover a wide range of constraint problem types. We piloted the selected problems to ensure that participants had sufficient time to explore them and to avoid those not easily understandable. Two of this article's authors independently rated the problems for solving difficulty on a scale of 1 to 5. We used the difficulty scale as well as the *type* of problem to balance the selection of problems for each participant. All groups addressed all problems the same number of times in aggregate. The final selected problems are: Word Crypto, Subset Sum, Sudoku, Scheduling, Magic Square, and Knapsack. Exact formulations are in the supplementary materials.

For each of the two problems assigned to them, participants had to carry out a visual modelling/specification task as well as a problem solving task, in sequence. Programmers in the CS and CP groups had to perform an additional programming task. The problem solving and programming tasks are not analyzed in this paper. Participants always completed all representation tasks first, which precludes bias due to the additional tasks performed by the CS and CP groups.

In the visual modelling/specification tasks the experimenter asked participants to try to illustrate a problem to a friend assuming that they can only communicate using paper. The experimenter also instructed the participants to try to use as few words as possible in the specifications. Our intention here was to prevent participants simply repeating or rephrasing the textual instruction given to them. Participants had 14 minutes to complete this task. Participants talked aloud, describing their thoughts and actions and, occasionally the experimenter would ask for clarifications or offered short reminders of the task. After the specification tasks they would complete the problem solving task and (if applicable) the programming task. Then they repeated the same process with their second assigned problem.

5.3 Apparatus

The experiment took place in a quiet closed room with the participant and the experimenter sitting at a table. Blank paper, pencil, pen, colored pencils, eraser, scissors and a pencil sharpener were provided. Two different cameras from two vantage points ensured full coverage of the paper as well as a complete view of the participant's actions.

5.4 Raw Data

The analyzed data consists of the two streams of video for each of the participants (a total of 801 minutes of video per stream), and the paper output from their specifications (available in the supplementary materials). Snippets from these materials in the remainder of this paper appear marked with the expertise group (Non-CS,CS,CP), the number of the participant within that group (from 1 to 10) and whether this was their first or second problem (E.g., CS 7,2).

5.5 Analysis Methodology

We analyzed both the artifacts from each participant (their sketches) and their video. As a preliminary step, we transcoded the two video streams to allow simultaneous viewing of the different camera angles. In a first analysis step, we analyzed the artifacts produced by creating an affinity diagram of common occurrences and general themes. We then iteratively coded the features that appear within the sketches using MAXQDA [78] initially and then using Microsoft Excel, refining the code books on each pass, following grounded theory techniques. Towards the end we settled into two main groups of codes: Visual Elements (VE) and Parts of Programming Language (PL). The categories were developed using a langauge based approach as this was most flexible. We also iteratively coded the videos for occurrences of gestures and, in a final pass we analyzed process elements (e.g., in which order did examples and generalizations took place). The authors meet 3 or 4 times during this period to clarify any ambiguities in the categories. These form the basis for the analysis in Sections 6.1 to 6.3.

5.6 Coding Validation Analysis

The bulk of the coding was performed by the first author. In order to ensure the robustness of the coding system, the remaining three authors performed two independent coding passes of a subset of 50 of the 230 artifacts in the first pass, and 25 of the 230 artifacts (3 out of 30 participants) and 6 of the 60 videos, at two stages in the development of the code books (approximately 20 person-hours of joint coding in total). We calculated the inter-coder reliability ratio as the number of agreements divided by the total number of codes in the Visual Elements (VE) category, averaged across all participant outputs (result: 94.5% agreement for the final coding session).[1] To account for randomness, we also calculated the Cohen Kappa statistic [49] using the scikit-learn python library [61] ($\kappa = 0.58$ when coding VE and PL as separate codes and $\kappa = 0.37$ when coding VE-PL pairs). The numbers reported above refer to the final coding validation only and roughly correspond with what is expected in a qualitative coding of this characteristics, especially taking into account that the Kappa coefficient also has limitations [76, 77]. Regarding the nature of the interpretation of participants' outputs see also Section 8.4. The supplementary materials contain the CSV and python code that we used for these calculations.

6 Analysis

The analysis results are split into three: the graphical elements (*signifiers*) that participants used; a catalog of problem constructs (the *signified*); and the relationships between elements in the previous two, with frequencies in which participants across groups mapped them as well as a summary of the most relevant regularities. Note that, although we provide multiple numerical measurements from the data (e.g., entropy), our analysis approach does not allow statistical estimates of the reliability of these measures; readers should exercise caution when applying or extrapolating these numerical findings.

6.1 Elements of Visual Representation (VE)

Participants created a variety of marks on paper to describe problems. Although we asked them to use graphical means to convey the problems (marks on paper, or paper cut-outs), we noticed early on that, to explain the permanent graphical elements and their relationships, most participants used also gestures. To avoid missing a potentially important source of meaning, we considered gestures in our analysis. From now on we refer to permanent marks on paper or physical objects (cut-outs)[2] and their characteristics (e.g., color) as *marks*, to distinguish them from *gestures* (e.g., pointing with a finger). Together, marks

[1] The data was coded from two perspectives, the Visual Elements (VE) perspective, looking at marks on the paper, and Parts of Problem Language (PL) perspective, looking at problem elements used in representation. See section 6 for more details.

[2] Although we provided participants with scissors and paper to create cutouts, only two of the CP participants used these. Moreover, these are straightforward to map to the other categories in this section. Thus we do not analyze cutouts as a separate representation.

and gestures are the *visual elements of representation* (VE), i.e., the graphical vocabulary to visualize problems.

The tree of visual elements in Figure 1.B contains the categorization of marks and gestures that emerged from the analysis. We made categories based on interesting regularities rather than following predefined classifications of marks such as Bertin's [7, p.44] or Munzner's [54, p.96][3] and gestures (e.g., [26, 40, 60, 65]), which might not have captured with sufficient detail some of the interesting phenomena in our specific scenario of problem representation, or might provide too many categories, making the analysis unnecessarily detailed.

6.1.1 Marks (VE1)

We observed a variety of *marks* and mark characteristics (visual channels). We divided them into eight categories for which we provide a brief description and a representative example from our participants' outputs in Figure 2. The categories are number-coded in Figure 1.B and the colors for the categories are reused in further figures.

VE1.1 - Graphical Containers. In this category we group elements that typically contain other elements inside. There are three sub categories: *boxes and circles* (VE1.1.1), *grids* (VE1.1.2), and *tables* (VE1.1.3). *Boxes and circles* are geometric shapes of a size large enough to fit text or other objects inside. These are common in real-world diagrams and were often used by our participants (21 out of 30 participants). *Grids* are different from tables in that position of a cell in the grid is spatially important (e.g., the top left cell might be special, or the adjacency of two cells or two columns carries a meaning), whereas in *tables* the order of columns and rows might be less important (rows and columns in tables are typically labeled and it typically does not matter in which order the columns appear on a table). *Grids* were used by 18 out of 30 participants and *tables* were used by 12 out of 30 participants. Seven participants used both grids and tables. These numbers are likely influenced by the problem assignments for those participants (some problems are already in a grid or table form).

VE1.2 - Symbols. This category refers to atomic graphical elements that are not labels or mathematical symbols (technically, labels and mathematics use symbols too but we separate them into their own categories). There are four subcategories of *symbols*: *punctuation* marks such as questions marks or exclamation marks (VE1.2.1); *people* (VE1.2.2), which are stick figures or people symbols not showing emotion; *emojis* (VE1.2.3), which are people symbols showing emotion; and *others* (VE1.2.4), which includes more abstract symbols such as ticks, small arrows[4], small geometric shapes (e.g., triangles, circles, squares, stars) and brackets. 25 out of 30 participants used *symbols*.

VE1.3 - Labels. Participants often used single letters or written words to represent objects or to annotate other elements on the page. *Labels* are different from *symbols* in that they are more directly connected to written language, and therefore usually have a pronunciation and could represent conceptual meaning or elements that are difficult to draw in a more straightforward way. Numbers are sometimes also used with *labels* to indicate order, otherwise the *labels* are usually unordered. Only 4 out of 30 participants never used *labels*.

VE1.4 - Text. We classify the use of textual language as *text* when it contains complete or incomplete sentences that go beyond just labeling. This typically happens when there is a verb. Note that participants were explicitly discouraged from using text directly in their representations, yet 11 out of 30 did. This is further discussed in sections 7.3 and 8.2.3.

VE1.5 - Maths. Participants used mathematical script in two roughly different ways. We separate *'simple'* (VE1.5.1) use of *maths* such as basic mathematical expressions like numbers and simple operators such as $+$,$-$,$>$,$=$,$\neq$, and Σ, from what we call *'advanced'* (VE1.5.2) mathematical expressions, which might include more complex constructions from set theory ($\forall$, $\exists$) or formal logic. 30 out of 30 used *maths*; this is likely to have been influenced by the problem types. Although maths is graphical, we will not refer to maths notation as graphical notation in this paper.

VE1.6 - Arrows and Lines. *Arrows and lines* are common elements in most diagrams and were also common in our data set. *Arrows and lines* are used to connect graphical elements on the page with each other (see – in Figure 2). 28 out of 30 participants used *arrows or lines*.

VE1.7 - Colors. *Color* is commonly used in visualizations for a variety of purposes, e.g., to show different instances or that two objects are the same. We made color pens available to our participants, which 20 of the 30 participants used to convey some meaning.

VE1.8 - Proximity. Sometimes participants put visual elements next to each other to indicate relationships between them. Although this is a more implicit type of relationship representation (there is not a permanent mark), implicit information might still be important. We coded instances where the *proximity* of elements was clearly used to convey meaning or when the participants mentioned the spatial relationship verbally. In some cases *proximity* works as an implicit version of the graphical containers subcategory (VE1.1 – e.g., grids where the grid is not explicitly drawn). We observed *proximity* encoding in 27 out of 30 participants.

6.1.2 Gestures (VE2)

To keep the analysis manageable we only categorized *gestures* with hands and fingers that interacted with the elements on the page or the page itself. These gestures took place when the participants explained the representations previously created on paper, were thinking about how to represent the problem, or during the process of writing or drawing. *Gestures* were sometimes complementary to marks on the page, when participants traced lines or circles already drawn on the paper, although they often did not correspond to marks on the paper. We distinguish between two types of *gestures*, *pointing* gestures and *path* gestures.

VE2.1 - Pointing. We put in this category *gestures* performed with a finger that highlight an object or area in the page. There are three subcategories: *serial* pointing (VE2.1.1), where the participant pointed at one object once or at multiple elements or areas successively; *parallel* pointing (VE2.1.2) where the participant would use multiple fingers to point simultaneously at several elements or areas; and *multi-tap* (VE2.1.3) pointing gestures where the same element is being repeatedly tapped. All 30 of the participants used some type of pointing gesture.

VE2.2 - Paths. *Paths* refer to *gestures* where the finger or hands trace a meaningful trajectory on the paper. We observed three subcategories: *drag* (VE2.2.1) where the finger starts on a page location, usually an object, and moves to a different place while still in contact with the surface; *manipulations* (VE2.2.2) which move a physical object from one location of the page or table to another (only applies to cutouts); and *lassos* (VE2.2.3) where the path traced is closed and delimits a regular or irregular area (usually with multiple elements inside). Due to their ability to connect to separate elements, *drag* paths are related to *arrows*. *Lassos* are also related to graphical containers because both categories can surround or contain other elements.

6.2 Parts of Problem Language (PL)

In this section we describe the problem concepts that participants represented, using examples from the knapsack problem described in Section 2. These do not refer only to the physical or conceptual objects in the problem but, importantly, also to the relationships between objects, the constraints that objects have to comply with, and how objects act or are acted upon. To arrive at categories that are descriptive of problem concepts in our data, we settled on a linguistic approach; our categories are analogous to the different *parts of speech* used in linguistic grammar analysis, but adapted to the specifics of graphical problem representation. Hence we call these *Parts of Problem Language*.

[3] For practical reasons we do not differentiate mark types and channels.

[4] Arrows that do not go from one place to another and instead indicate change (e.g., increasing or decreasing quantity).

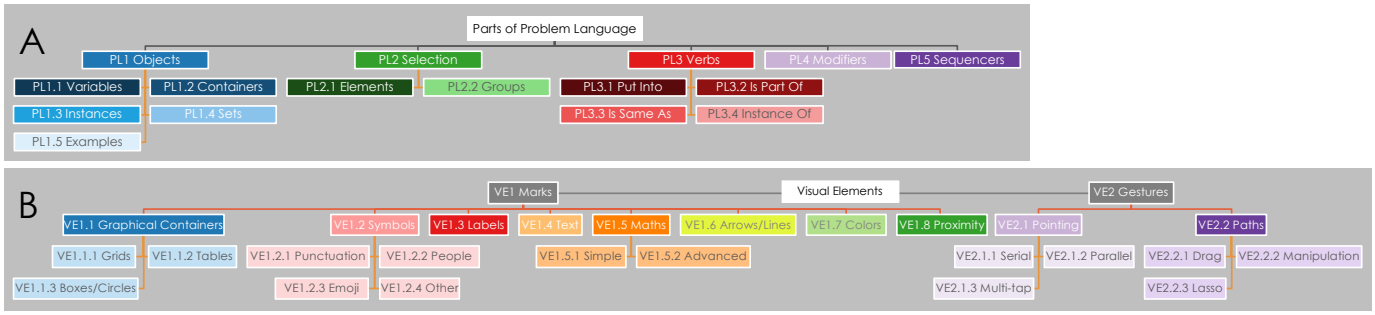

Fig. 1. Tree diagrams of the hierarchy of parts of problem language (PL–A,top) and visual elements (VE–B,bottom).

Note also that we chose a classification that expresses a broad range of constructs in problem description, regardless of whether those actually belong to the set of constructs used in actual constraint programming languages or other type of programming languages. For example, constraint programming languages are generally declarative, but our classification can describe procedural constructs (e.g., "put these objects in these variables, check that their sum is less than 10, if it is..."). Table 1 shows a glossary of approximate correspondences between our categories and the concepts and terms used in linguistics, programming languages and constraint programming languages.

Table 1. Correspondences between PL, CS and CP concepts.

ID	Concept	Part of speech	Programming	Constraint Programming
PL1	Objects	Nouns, Noun Phrases		
PL1.1	Variables		Variables	Decision variables
PL1.2	Containers		Data structures, collections	Collection of variables
PL1.3	Instances		Objects or Structs	Values
PL1.4	Sets		Types, Enums	Domain
PL1.5	Examples		Values, Data structure state, assignment of values to variables	Instance assignment
PL2	Selection	Pronouns, demonstratives		Indexing (applied to data), constraints (appl. to variables)
PL2.1	Elements		Indexing, Aliases	
PL2.2	Groups		Slices	
PL3	Verbs	Verbs	Properties, inheritance	Indexing (appl. to data), constraints (appl. to vars)
PL3.1	Put Into		Assignment	
PL3.2	Is Part Of		Attributes (and problem decomposition)	
PL3.3	Is Same As		Equivalence	
PL3.4	Instance Of		Instantiation	
PL4	Modifiers	Adjectives	Conditional Expressions	Constraints
PL5	Sequencers	Temporal Adverbs	Logical flow	

PL1 - Objects. The *objects* category is analogous to nouns or noun phrases in language grammars. Using the knapsack problem, each unique *object* (e.g., water bottle) would be an *instance* (PL1.3). *Variables* (PL1.1) are references to *objects* where the referenced *object* might change over time. For example, the *variable* `totalObjectsValue` might refer to the sum of the value of backpack objects at a particular time, which might vary during the process or for different solutions. *Containers* (PL1.2) are collections or groupings of *instances* or *variables*, roughly equivalent to the concept of data structures in programming languages. For example, a list of all the objects contained in the knapsack can be a *container* of several *instances* (e.g., water bottle, raincoat, sandwich) or of several variables (`object1`, `object2`, `object3`, etc.) if the objects have not been specified. *Sets* (PL1.4) are groupings of unique *objects* that denote the possible *instances* that can be referenced to by a *variable* or *container*. This is roughly equivalent to the concept of *Enums* in common programming languages. In our example, a *set* could be all available objects to put in the knapsack. When a *variable* or a *container* is shown in a particular state, we call that an *example* (PL1.5). E.g., a possible list of objects contained in the knapsack will be an *example* if it is represented as having four specific *instances* of concrete objects. Examples can also be negative, describing a state that is not valid.

PL2 - Selections. *Selections* refer to the highlighting of one or more *objects* which will be used with a *verb* (PL3, described below). There are two types of *selections*: *element selections* (PL2.1) and *groups* (PL2.2), depending on whether one or more *objects* are being selected. A description of a problem might use selection to denote, in our example, that a particular group of *objects* make up the *set* of objects that can be placed in the knapsack.

PL3 - Verbs. *Verbs* are actions or operations that are applied to *objects*. They include *put into* (PL3.1) which assigns an object or selection to a *container* or *variable*, *is part of* (PL3.2), which describes when *containers* are split into sub-containers to show a sub-part view, *is same as* (PL3.3) which indicates that two representations refer to the same *object*, and *instance of* (PL3.4) which selects an *instance* out of a *set*. An example of *put into* for the knapsack would be assigning a particular *set* to fit into the knapsack *container*. If the same object is represented twice, e.g., once to describe what is currently in the knapsack and once in a list of objects ordered by value, then a mark indicating that both are the same would be functioning as *is same as*.

PL4 - Modifiers. *Modifiers* is a large category covering all the constraints that can be applied to *objects*, *selections* or *verbs*. *Modifiers* can limit values, indicate that values should be all different, to maximize or minimize or to find all possibilities. For example, a representation that indicates a variable that contains the sum of all weights in the knapsack has to be below 30 would be considered a *modifier*.

PL5 - Sequences. A mark or a gesture functions as a *sequence* if it provides an indication of temporal order, akin to a temporal adverb or adjective in linguistic terms. *Sequences* were rarely used but some participants implicitly or explicitly provided procedural descriptions (e.g., algorithms). An example sequence in our running example would be an indication that the list of knapsack objects has to be populated first with the object of highest value, then with the second, etc.

6.3 Semiotic Mappings

We investigated the mappings between visual elements and parts of problem language that participants created. This is a basic form of semiotic analysis that matches signifiers (in our case VEs) with the signified (PL). The results are provided as CSVs in the supplementary materials and summarized in Figure 3.

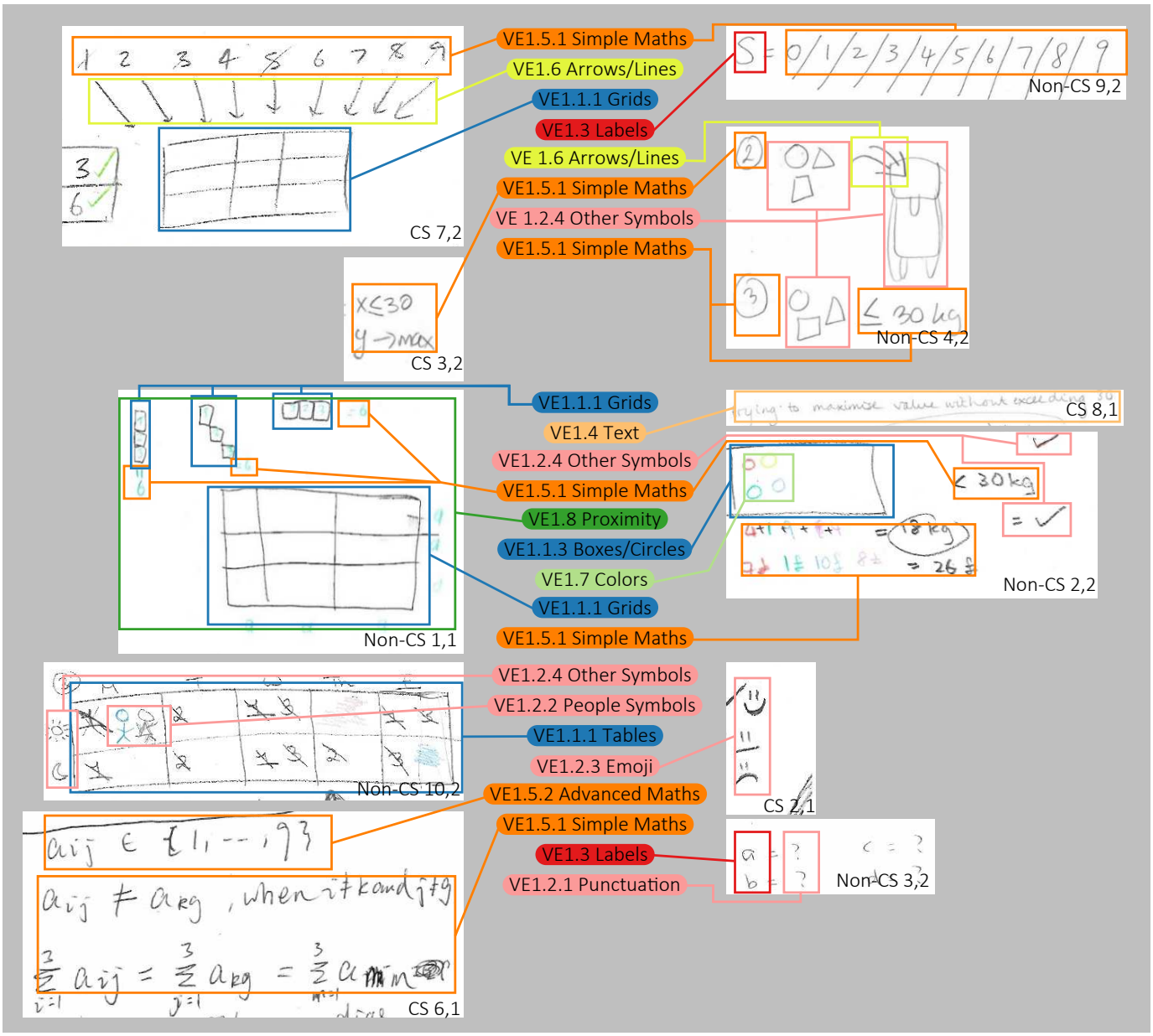

Fig. 2. Snippets of participant outputs highlighting examples of all visual marks.

6.4 Input/Output Entropy Analysis

If the same type of visual element is used to carry out many different functions (e.g., if we use textual *labels* to represent *instances*, name *sets*, label *examples* and tag *modifiers*) this might make it harder for the reader of a diagram to recover the meaning of the label when they encounter it. To quantify this we can calculate the Input/Output Entropy of the mappings. In other words, we can calculate how many bits of additional information we would need to recover which PL, a VE is referring to. We perform this calculation using the finer categories in the VE tree, and apply the following formula to the aggregated mappings of each expertise group and the total:

$$\sum_{o=1}^{m}\left(\sum_{i=1}^{n}\left(-\frac{x_{io}}{\sum_{i=1}^{n}x_{io}}\cdot\log_2\left(\frac{x_{io}}{\sum_{i=1}^{n}x_{io}}\right)\right)\cdot\frac{\sum_{i=1}^{n}x_{io}}{\sum_{o=1}^{m}\sum_{i=1}^{n}x_{io}}\right)$$

where i is the input (PL) index, o is the output (VE) index, x_{io} is a particular number of uses for that input and output combination, m is the total number of outputs and n is the total number of inputs.

Table 2 summarizes the results, and shows how the higher the level of formal education in modelling, the smaller the entropy value. It should be highlighted that, in our calculations we added the rule that the use of maths would not add entropy. The reason is that maths can be effectively used for several purposes, using well established symbols that would not add confusion (e.g., brackets for sets, numbers as instances). Although it would be possible to apply the same logic to other areas of the VE tree (e.g., arrows could be combined with labels or colors to separate different functions), we did not observe occurrences of this.

Table 2. Input/Output Entropy for the different cohorts, in bits.

Cohort	Non-CS	CS	CP	Total
Input/Output Entropy	1.26	1.23	1.12	1.31

7 Findings

This section collects the main insights gained from the analysis presented above. Insights are discussed by topic and highlighted bold.

7.1 Participant diagrams are chaotic but there is regularity

One goal of our study was to understand the consistency and level of sophistication of the graphical means that different groups use to

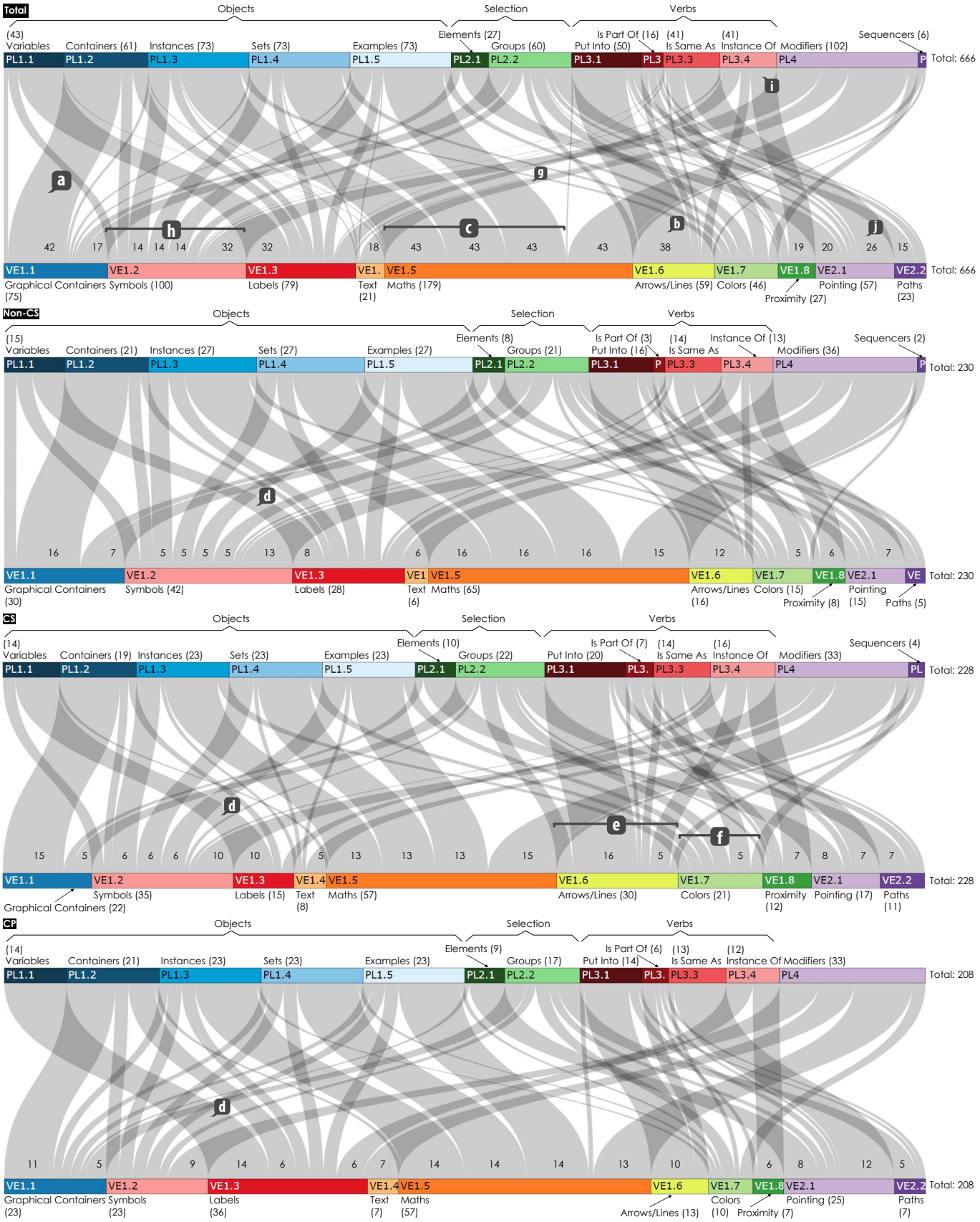

Fig. 3. Sankey diagrams showing the mapping between VEs (top) and PL (bottom) for All, Non-CS, CS, and CP participant groups.

represent constraint problems. A priori, results could have varied from a perfectly chaotic set of graphical representations (the same marks or gestures are used inconsistently to represent any concept), to a perfectly defined and unambiguous language, almost a graphical programming language (people are perfectly consistent and use the same marks always for one and the same purpose without any space for ambiguity).

The findings of the study, especially in the form of the diagrams in Figures 1–3 and the entropy Table 2 show a picture somewhat in the middle. **Participants' representations of constraint problems are not consistent** (input-output entropy values are not close to zero) and substantial variation exists in how participants represent the same PLs. For example, *variables* (PL1.1) were represented by participants using *graphical containers*, *symbols*, *labels* and in one instance *maths*. Very diversely represented PLs are group *selection* (PL2.2 – 7 different first-level marks or gestures at the high level to represent *selections*: *graphical containers*, *symbols*, *arrows/lines*, *colour*, *proximity*, and both *pointing* and *path* gestures), and *instances, sets and examples* (PL1.3-5 – 6 types of marks). However, **neither are the mappings completely void of regularity**; if this was the case the calculated input-output entropy values would not be in the 1.1–1.4 range that we observed (Table 2) and would instead be closer to the theoretical maximum given the number of categories in our scheme—3.7 bits.

7.2 Intrinsic, Expected and Unexpected Regularities

We observed many regularities in the mappings. Here we present the most obvious in three loose groups based on whether they are *intrinsic* to the nature of problem language and marks, *expected* due to the setup of the problem or the makeup of the participant groups, and *unexpected*.

7.2.1 Intrinsic

We saw that ***graphical containers*** **are mostly used as** ***containers*** (42 out of the 61 uses of *containers*) (Figure 3, a). *Graphical containers* (VE1.1) are, by definition, able to contain other elements inside, and therefore their use to represent *containers* (PL1.2) is a natural choice for most people (see Figure 4.A). In other words, *graphical container* marks and the concept of *containers* perform very similar functions in the representation and the conceptual sides of the problem respectively, and this leads participants to this pairing.

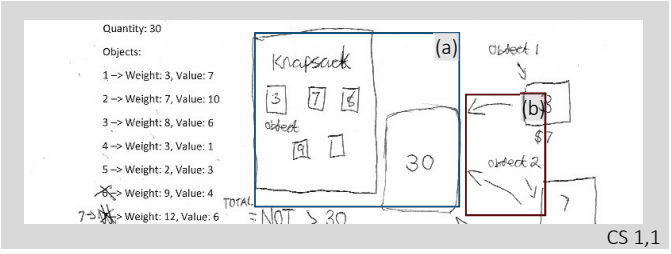

Fig. 4. Snippet of artifact showing (a) *containers*, (b) *put into*.

In a similar way, ***verbs***, which in language usually connect a subject with an object seem **easier to represent through** ***arrows/lines*** which connect two separate elements in the graphical space (Figure 3, b). A clear example of this is *put into* (PL3.1), which is represented largely by *arrows/lines* (38 out of 50 cases–see Figure 4.B). We could describe this as the shape of a visual element determining the syntax of the visual language.

We also found that the **use patterns of** ***drag*** **gestures** (VE2.2.1) **are parallel to those of** ***arrows*** (VE1.6). This makes sense, since *drag* gestures replicate the characteristics of *arrows/lines* (they connect two separate elements), except that they are transient visual elements rather than permanent marks. Other similar correspondences between gestures and marks exist: ***lassos*** (VE2.2.3) **sometimes work as** ***boxes/circles*** (VE1.1.3) and ***serial*** **and** ***parallel*** **pointing gestures** (VE2.1.1 and 2.1.2) **often do the same work as** ***lines/arrows***.

The paragraphs above describe regularities that are likely a consequence of the intrinsic characteristics of the VEs and their relationship to the PLs; however, we note that there are always representation alternatives, even if most participants chose not to use them. For example, containment can be represented via *arrows*, *verbs* can be represented by containment; both can also be expressed with *text*.

7.2.2 Expected

Numbers, which appear in our scheme as *simple maths* (VE1.5.2) **were often used to represent** ***instances***, ***sets*** **and** ***examples*** throughout the different representations and groups (Figure 3, c). This is likely due to the problem set selected for the experiment being often number-based.

We also saw differences in patterns across the participant groups that can be expected from their formal training. **CP used more** ***advanced maths***, probably because they are familiar with the advanced notation and regularly use it. Due also to differences in training, the mappings also show a clear preference by the **CS and CP groups for using** ***labels*** **to represent** ***variables*** (as they usually do when programming with editors) (Figure 3, d). In the case of CP, *labels* were the only mechanism to represent *variables*, whereas CS used also *symbols* and numbers[5]. Some **non-CS also used unlabeled** ***graphical containers*** **to represent** ***variables*** (i.e., a box that contains a different value at different times). Although this is a common metaphor in beginner programming textbooks to introduce the concept of variable, we only saw this in three problems by two participants.

7.2.3 Unexpected

Some trends are not easily explainable. We noticed that **CS use** ***arrows/lines*** **for more purposes than other groups** (for *groups*, *put into*, *is part of*, *is same as*, *instance of* and *sequencers*, whereas non-CS only used them for *groups*, *put into*, *is part of* and *instance of*, and CP used them only for *put into* and *is part of*) (Figure 3, e).

CP used graphical containers to represent more PLs than any of the other groups (CP: *containers, instances, sets, examples, element selection and group selection*; CS: *containers, element selection and group selection*; Non-CS: *variables, containers, element selection and group selection*).

The Sankey diagrams also show that **CS used** ***color*** **more often than other groups** (for 21 PL-VE problem instance pairs vs CP 10 pairs and non-CS in 15 pairs) (Figure 3, f).

7.3 Use of Text vs. Graphical Marks

Our experimental protocol included asking participants to minimize the use of text and written language and favor the use of graphical means in their representations. Despite of this, we observed the use of ***text*** (VE1.4) in the form of sentences **in a substantial number of occasions** (21 problem instances across all expertises). Participants used *text* as a last resort for elements that they found important but difficult to represent otherwise. Therefore, *text* use provides indirect evidence about which PLs are harder to represent visually. For example, CP Participant 6,2, when solving the subset sum problem, encoded the constraints through *text* in the following way: *"Find All Subsets Which Add To Zero"* (Figure 5.

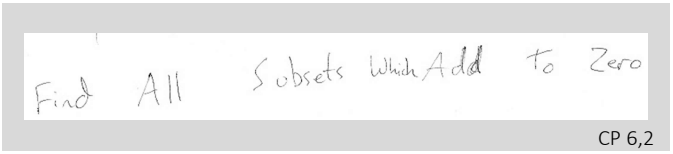

Fig. 5. Snippet of artifact showing text use.

The Sankey diagrams and the tables in the supplementary materials show that **text was mostly used to represent modifiers** (PL4), **with a few examples of use for instances, sets and examples** (Figure 3, g). In other words, participants found it difficult to represent constraints with any other types of visual marks, including *maths*.

7.4 Use of Maths vs. Graphical Marks

The **generalized and consistent use of mathematical notation** (*maths*–VE1.5– is the most commonly used group of marks) is noteworthy. Some concepts such as addition, or inequality comparisons (greater than or less than) are plausibly quite easy to represent through graphical constructions. For example, a *greater than* comparison is visually perceived by a visually obvious difference in height when

[5]Using only numbers to denote a *variable* only happened once in our dataset, and is an example of a relatively easy to avoid mapping for *instances* and *variables*—numbers are often used to designate specific values because they have a fixed meaning, rather than to name elements that can change.

represented in a bar chart, or even by the position of an element in a number line [41]. Yet when representing the concept of *greater than* the overwhelming majority of our participants chose to simply write a mathematical expression that uses the symbol >. Mathematical language is a formalized language that is used to represent reality and that is learned by most people early in their education; therefore it is not altogether surprising, although still significant, that our participants chose to **use mathematical notation instead of drawing from visual properties or inventing their own graphical notations**. The point also extends to the use of numbers—although there might be other representations of cardinality and ordered sets that might be visually more evident, people's use of numbers and digits is second-nature. These points are further elaborated in Section 8.

7.5 Use of Symbols vs Labels

We separated *symbols* (VE1.2) as a separate category of graphical marks. *Symbols* are similar to *labels* in that they are very flexible marks that can represent many PLs, and be quite abstract. **Participants used *symbols* to represent specific *instances, variables, modifiers* (constraints) or to refer to *containers*** (see (Figure 3, h)). Just as *labels*, they often provide a shorthand to refer to other elements on the page that might otherwise be time consuming to redraw (see Figure 6).

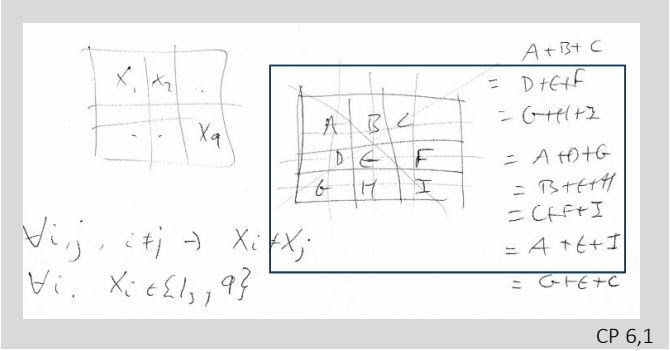

Fig. 6. A *label* used to avoid redrawing the grid.

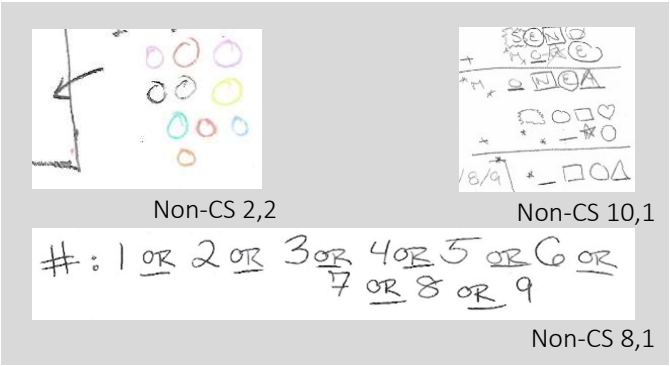

Fig. 7. *Symbols* representing an *instance* (l), *variable* (r) and *set* (below).

Despite the similarity of functions with *labels*, ***symbols* were** still **heavily used**. Figure 7 shows the use of *symbols* for a *instance, set and variable*, demonstrating their versatility (*symbols* were used by 25 out of 30 participants and *labels* were used by 26 out of 30).

7.6 Implicit Representations

Not everything is represented using marks on the paper. **Participants often represented elements, especially *verbs*** (PL3), more **implicitly either through proximity on the page or gestures**. This is important because implicit graphical relationships and gestures can be difficult to recognize by human and computer interpreters (this is further discussed in Section 8.2.5). Some examples of the most common implicitly represented elements follow.

Participants often indicated implicitly that something is an *instance of* (PL3.4) **through *proximity* on the page** (VE1.8–18 out of 27 times–Fig 8.a) (Figure 3, i). Similarly, ***same as*** (PL3.3), **is often signalled by *pointing*** (VE2.1–26 out of 57 times–Fig 8.b) (Figure 3, j). Interestingly, **CP did not make *same as* explicit in a graphical way.**[6]

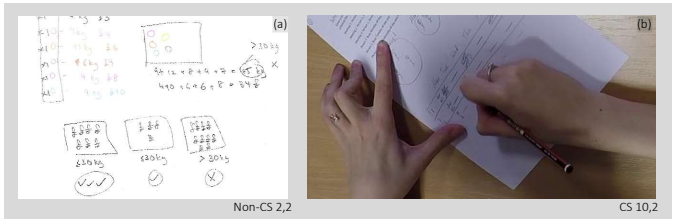

Fig. 8. Figure showing (a) *instance of* through *proximity* (b) *same as* with *parallel pointing* gesture.

Another part of problem language that is **not commonly represented on paper but appears sometimes as gestures is *element selection*** (PL2–Fig 9), especially for CS and CP cohorts (39 times overall, out of which 15 times for CS and 15 times for CP). Participants also denoted *is part of* through gestures (5 times–Fig 10) and *proximity* (5 times–Fig 11). Interestingly, CP did make *is part of* explicit for the most part (4 out of 6 times).

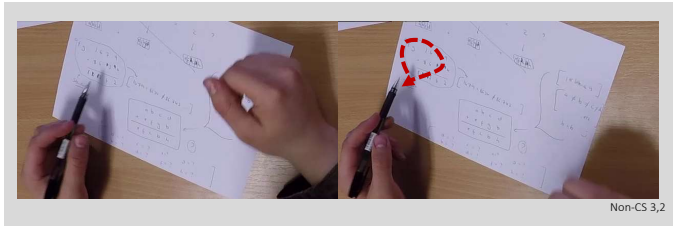

Fig. 9. *Element selection* represented through *lasso* gesture.

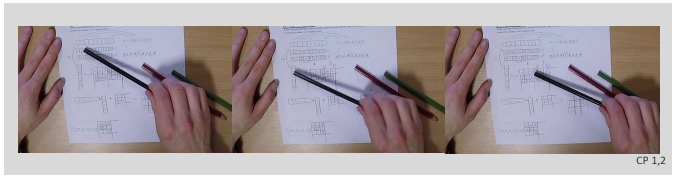

Fig. 10. *Is part of* represented through a *serial pointing* gesture.

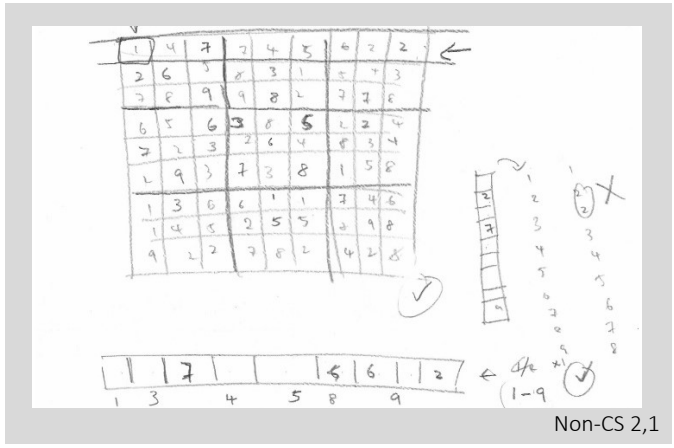

Fig. 11. *Is part of* represented through *proximity*.

Finally, looking at the Sankey diagrams from the bottom-up, **implicit representation elements** (*Proximity and Gestures*, VE1.8 and VE2) **are only used to represent *verbs* and *selections***.

7.7 Bottom-up vs Top-down Processes

We were interested in understanding not only the kind of mappings that people make, but also how the process of creating representations

[6]CP establish the *same as* relationship by naming objects with the same *label*, which is explicit but we do not consider graphical. See also Section 8.2.3.

takes place and may support understanding of the problem. For this purpose we carried out an initial process analysis by looking at the videos of participants creating their representations. More specifically, we looked at when they created examples as opposed to when they described general structures or relationships.

The analysis revealed that **most participants** (all but one) **start trying to describe the problem with general rules and abstract structures**. **Participants then resort to examples that often expose the incompleteness of their specifications** (26 out of the remaining 29), and **finally they review their output to establish relationships between the general and the specific**. Of the three participants that stayed at the abstract level without examples, two were CP and one CS.

An additional observation about process is that **participants often redraw parts of their representations**, perhaps to clarify and clean their output, but perhaps also to give themselves the time to understand, review, and debug what they have done as part of their thinking process. Ten out of the thirty erased or crossed out parts of their previous marks.

8 Discussion

Here we interpret the findings from Sections 6 and 7 and elaborate on their implications for the goals stated in Section 3.

8.1 Addressing the Research Questions

Our analysis of the data from thirty participants provides a picture of the visual elements that they used (Q1), and the kinds of constructs that they aim to represent (Q2). These are represented as the VE categories tree in Figure 1.B and the PL categories tree in Figure 1.A respectively. The bulk of our results, however, is the description of the ways in which VEs are used to represent the PLs by the different groups (Q3). These are summarized in the diagrams in Figure 3 which are then dissected and complemented by examples in Section 7. The following discusses the findings around the most important topics.

8.2 Key Topics

8.2.1 Towards a consistent problem graphical language

In *Visual Language, Global Communication for the 21st century*, Horn argues that visual forms of communication "have begun to encounter one another and integrate into a larger, more inclusive language" [31, p. 5]. The question occupying us is whether this "confluence visual language" seems to be happening, and whether it applies to the specification of constraint problems. Within this chosen area of interest, our evidence suggests that the use of graphical means to represent problems is not consistent across people and that people only partially know how to take advantage of graphical means to represent problems effectively. More formal modelling expertise translates into slightly improved consistency (Table 2).

Nevertheless, the found regularities provide a starting point for the design of languages for constraint problem specification. We can leverage this knowledge in three ways: first, designers can choose graphical representations, concepts and mappings that are somewhat familiar or "natural" to people, with the associated advantage of making the language easier to learn and more straightforward; second, when people's current representations fail to be complete, accurate, or readable, designers can use this information to design better alternatives; and third, knowing which concepts are hardest to represent highlights what will require better solutions and more training.

8.2.2 Diagrams vs Maths

One might claim that we already have an appropriate language to represent problems: the language of mathematics. This is somewhat corroborated by our analysis; we found examples in our data that indicate that maths notation does already much of the work of representing problems. A basic example is representing ordered sets. It is difficult to beat numerals (which we classify as *maths*) to express order (our participants all used numbers when trying to represent order). We also observed that participants very rarely use graphical means to represent number comparisons, instead using mathematical symbols (e.g., $>$, $<$). Although these symbols are not completely abstract or arbitrary (see Figure 12.left), there exist graphical alternatives in visualization to represent differences and comparisons (e.g., Figure 12.right) yet people generally still prefer the short hand.

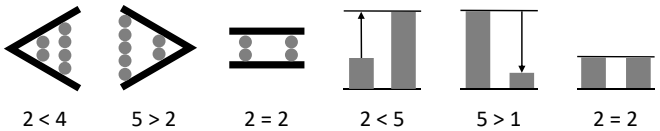

Fig. 12. The greater than, lesser than and equals symbols can be considered shorthand for visual representations of comparison (left); other graphical representations of comparison are also possible (right).

Mathematical notation is powerful enough to represent most problems, and its symbols are mostly consistent and precise [86]. However, translation into maths is often difficult, time-consuming to write and comprehend and does not always fit the problem. For example, geometrical axioms are easily representable through mathematical equations but a simple diagram can be faster to understand and remember. Whether illustration and graphical representation enriches or detracts from maths is a debate of philosophical consequence out of scope for us here (see [59]). We make the assumption that graphical representation can facilitate people's understanding of problems and data. The existence of early illustrations in science [42], and of InfoVis as a field can be considered a form of support for this position.

Our findings suggest that forcing people to use new graphical representations for elements that are already well assimilated from maths, as in Figure 12, is probably unnecessary and, at worse, counterproductive; yet restricting ourselves to maths only is not a good option either, for the reasons in the previous paragraph, and because there are likely cognitive benefits that escape current mathematical representations. Among these are the benefits already highlighted by Margaret Burnett and others in the long history of the study of visual languages (e.g., [12, 13, 50, 55]), such as the higher dimensionality of visuals, which can lead to better concreteness, directness, explicitness and feedback. We also discuss visual indexing as another possible advantage in the next section.

8.2.3 Containers and Symbols vs Labels

The choice between a (textual) *label* and a *graphical container* to represent a content abstraction (e.g., a *variable*) has interesting consequences in terms of visual language: a *label* is a referent that exists in the domain of language (a name), whereas a box is a referent that exists mostly in the domain of space (the page). Both accomplish the same kind of work (a form of abstraction as defined in [35,52]: a way of representing multiple elements by hiding them behind a single referent) yet they require very different types of retrieval. In the case of a box, when someone needs to go back to it, they can remember the approximate location on the page and its shape/size/color, or follow arrows from where they are currently looking (if they know they are connected). Conversely, a text label can retrieved by phonological or conceptual memory, or by pattern matching the label text within the content of the page to find other references to the same label. We suspect that the different ways of retrieving elements can have important repercussions for the building and reading of these diagrams because the retrieval mechanism is a key part of cognition itself [5]. However, the exact impact and importance of retrieval within the representation process of is not well known and requires further study.

Unlike graphical containers, *symbols* were used often in all expertise groups and for non-CS and CS more often than *labels*. Since text-based *labels* and *symbols* can fulfill the same functions, why do participants use *symbols* instead of only *labels*? We see three possible advantages:

First, if *symbols* are only used to represent one concept, one part of problem language, or even a subset of elements, recognizing the symbol provides straightforward association to its function in the diagram. If instead we use *labels* for virtually anything (e.g., to designate variables, containers, instances, sets), the reader will have to retrieve the function based on cues and context. For example, the author/reader of the diagram may have to remember the function of that object, or derive it from its use in context/syntax (e.g., a *container* cannot *be same as* a *variable*), or from another hint in the meaning or form of the *label* itself (a capital letter convention, or the meaning of the label, such

as in `matrixKnapsackObjects`). This is often the case in text-based programming languages, and is sometimes addressed through color and fonts automatically changed by syntax highlighting. In our data set we did not find any instance of the use of color or any typographic (calligraphic) parameter for the purpose of differentiating PLs. Another example of the use of special symbols to designate function is in *maths*, where vectors are often marked with an arrow on top, or the use of Greek letters, which might denote a particular type of variable (e.g., angles). Note that these graphical distinctions are not strictly necessary, but might provide cognitive advantages.

Second, *symbols* are special types of glyphs, some easier to find on a visual search due to pre-attentive processing [84, p.152], [85]), as opposed to uniform-looking text *labels*. Although one could argue that visual search does not matter when specifying a problem description, we believe, from our observation of the participants, that going back and forth between the different areas of the page (or pages) is common enough for this to have an impact (see also arguments that present marks on paper as a key part of the cognitive process [5, 15]).

Third, participants might like representing some objects in a more literal way. Turning a *symbol* into an icon (e.g., using a stick person figure to represent a person in the problem) establishes a more direct link between the significant and the signifier (see, e.g., Non-CS 10,2 in Figure 2). This, in turn, might facilitate thinking about the problem by bringing it closer to a familiar situation through analogy [22].

It is important to remark that the use of *symbols* to denote a particular type of elements in the page also has an input and cognitive cost that increases with the number of special symbols or categories of symbols used. Most ad hoc symbols will take longer to create and to redraw (the nice simple shapes are, for the most part, already taken by letters and maths) and some energy to remember and keep consistent.

This might be the reason why we found that the CS and CP groups used *symbols* less, with three CP participants never using symbols. People with a more formal expertise in the programming area are constrained to labels in their daily practice, and might have developed strategies to cope with the ambiguity.

8.2.4 Graphical Abstraction, Deduction, Induction and Solving

During the analysis it became evident that one of the difficulties of visual representation is that abstraction and some actions and relationships are hard to represent graphically. For example, it is hard to find effective visual representations for a structure (e.g., a *grid*) that has a variable number of columns or rows. A similar problem appears when trying to represent some constraints (for which our participants often resorted to *text*), or when representing procedures, sequences, repetition or recursion. This does not mean that it is impossible to graphically represent those (e.g., Figure 13). Stenning and Oberlander [73] discuss a similar problem in the context of Euler diagrams; graphical representations make complex structures concrete and explicit, which makes them easier to acquire and process, yet this involves a tradeoff with the ability to represent multiple alternatives through abstraction.

$$\begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix}$$

Fig. 13. Example of variable size representation in a matrix grid.

In 7.7 we described how participants generally start with the abstract and general parts of the problem representation and then become specific (examples), often going back and forth. Although our observations are undoubtedly affected by how problems were formulated in our empirical design, the evidence suggests that processes are not exclusively bottom-up or top-down, but a combination of the two, often with several iterations. In other words, it would be inaccurate to assess the process that participants go through as entirely inductive or deductive.

Anecdotally, we have seen this when teaching constraint programming. Constraint programs (*models*) are very abstract; before one can get to a final model of the problem one might need to understand specific cases (this is particularly true for novices). Thus we suspect that, although most current tools to solve constraint problems seem appropriate for describing the final formulation of the problem, they may not be ideal to support the process of building and understanding it. Tools that support people's complex and non-linear thought processes have the potential to increase their effectiveness, especially when learning. This "constructive" approach showed benefits in other areas of visualization (e.g., [32, 51]).

A related issue is description vs. solving of problems. Although textual constraint programming languages are considered declarative (as opposed to imperative), we observed that many participants describe problems procedurally (e.g., CS 2 in Figure 14). This might be an important source of friction between how people naturally think about problems and how most current constraint programming languages work, and therefore an opportunity to improve constraint modeling learning and make it more accessible. Nevertheless, we note that constraint programming is inherently declarative and integrating procedural elements requires further research in constraint solvers.

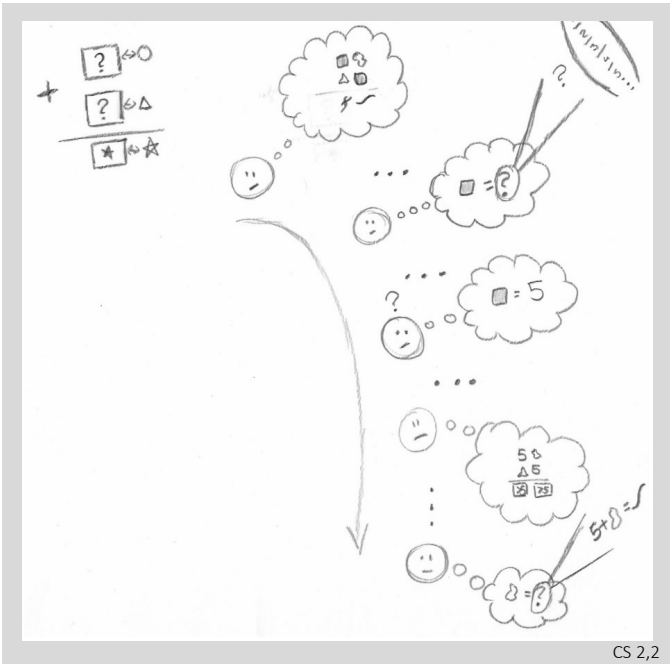

Fig. 14. *Word crypto* solved through procedure: define *containers* and *instances*, assign *variable*, define *instance*, consider next value.

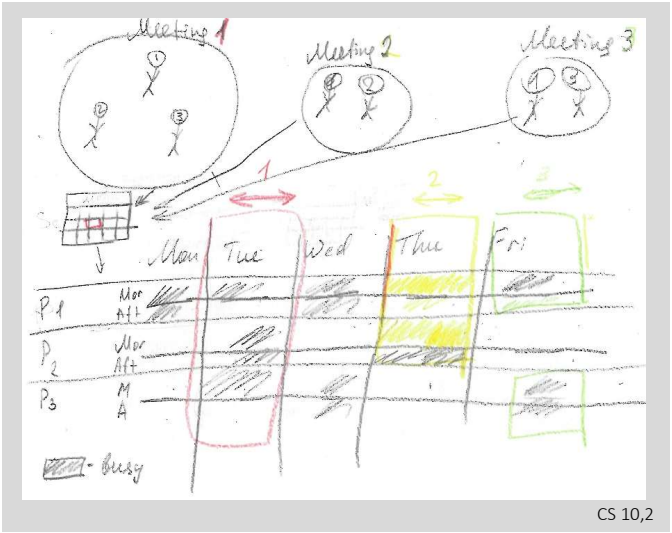

Fig. 15. Ambiguous graphical formulation for the *Scheduling* problem. The relationship between the groups being scheduled and the two representations of the calendar could be interpreted in different ways.

8.2.5 Under-specification and Implicit Information

Perhaps partly due to the difficulty of representing abstraction and actions graphically (Section 8.2.4), we observed that people often under-specify the problems (e.g., the representation in Figure 15 can easily be misinterpreted). Although sometimes people simply do not provide enough information to unambiguously describe a problem, other times the information is there, just not committed to written marks. Participants often used gestures to indicate relationships between items or actions. There is evidence that gestures can be key constituents of the cognitive process or a representation thereof [24,36,44,45]. From other domains we also know that sometimes subtle gestures can contain very relevant information (e.g., [39]). Therefore, it follows that interfaces might want to exploit gestures and other sources of implicit information (e.g., proximity in the page) to support the process.

Another conspicuous source of implicit information is temporal order. Looking only at the participants' final results, the representation is difficult to understand. However, watching the specification process (e.g., by playing the video) is easier because the order in which things are specified contains information; people tend to assume that the information available to them during the process is the same as that the reader will have, and fail to see that further modification and addition of elements can make the specification ambiguous or harder to read at a later time. We call this *process state bias*: people assume that the state at creation time will be the same as at reading time. This affects 2D media more because objects in the page can be added anywhere in the space, as opposed to textual media where the order is assumed to be left-right, top-bottom. This is connected with provenance representation (e.g., [27,66]).

Relatedly, participants often fail to explicitly mark the elements of the problem that need to be solved, minimized or maximized (the final outcomes), to differentiate them from intermediate steps or even from sketches or annotations in the page that were useful during specification. This is relevant because the additional information, similarly to the previous paragraph's argument, might obscure the representation, but also because constraint solvers might be able to optimize the process of finding a solution by relaxing or changing intermediate representations. In constraint programming languages the sought solution is usually indicated explicitly (e.g., Essence+ uses the reserved word *find*).

8.3 Suggestions for Designers

Here we distill the findings and discussion into suggestions that might be useful for designers of constraint problem specification (CPS) visual languages, regardless of whether these are for communication with others, oneself at a another time, or for computers to solve (e.g., as part of a constraint solver user interface).

- CPS languages should probably be hybrids, combining different forms of expression drawn from mathematical notation, text, iconic language and others:
  - Leverage common knowledge of basic mathematical notation, even if "more graphical representations" exist.
  - Consider adding text descriptions for input/output [37].
- Support the process rather than just the outcome:
  - Consider that problem specification is not only top-down or bottom-up, but an alternation of the two.
  - Intermediate states might be wrong or under-specified but likely part of the natural process of specification.
- Consider leveraging implicit information as a source of data:
  - Gestures can show useful information not written as marks.
  - Proximity and temporal information also encode useful information for reading a problem specification.

8.4 Limitations and Future Work

Readers should be careful not to interpret the regularities we describe in Sections 6-8 as a direct blueprint of how this kind of visual language should look like. There are two main reasons. First, what participants did might not be the optimal way to describe a problem, could be incomplete or even contradictory—the design of a visual language has to balance "naturalness" and ease of learning with other aspects such as correctness, completeness and consistency with the domain the language is being applied to. Second, if the language is to be read by a machine its design should support fast and accurate parsing, which is not trivial (see [46]). Conversely, the ontology of the language and the forms of input for the UI (e.g., dragging elements from a palette or using templates vs. enabling free pen input) could have a substantial impact in how easily the language is to be understood and put to use by humans (we know of this kind of effect in InfoVis [51]).

Likewise, our quantitative calculations for Entropy use non-trivial maths, which might make them appear deceptively precise. Although these measurements are useful to get an idea of the level of consistency in the use of VEs, they are nevertheless very dependent on the number of categories and their coding, which are both somewhat subjective and affected by the skill of the experimental analyst. Additionally, these measurements are relative and therefore should never be compared to entropy measurements with a different coding schema.

Our entropy analysis is partly based on the assumption that it is better to have a different PL represented with a different VE. In Section 8.2.3 we argued for the likely advantages of this approach. However, it might be possible to use the same category of VE for multiple functions in a way that preserves these advantages; specifically, a visual language could use *combinations* of visual elements such as labels+arrows, symbols+arrows, or colour+containers to generate a much larger set of sub-classes of visual elements that can then be mapped one-to-one with the PLs, perhaps even in a more granular way (e.g., by splitting *PL4–Modifiers* into different types of constraints and qualifiers). We did not observe our participants using this approach and therefore we can neither recommend nor discourage its use. This ability to articulate visual elements can be very expressive and takes the visual language closer to the versatility of spoken/written language but more research is needed to discern whether this will have unwanted consequences in the learnability, writability and readability of the language.

In this paper we have looked at the generation side of the problem: participants produced problem specifications to the best of their ability. However, we now know less about which of these descriptions tend to be more complete, accurate and interpretable by humans. Future work should address this by reversing the question of the analysis to something like *what patterns of representation used by participants tend to be most complete/accurate/interpretable by humans?* The dataset of participant-generated diagrams included as part of the supplementary materials can be a starting point. Similarly, deeper semiotic analysis of our data is possible. For example, looking at the implied syntax of the participant sketches might reveal interesting patterns of how and why certain types of visual elements connect to each other.

A decision of our empirical design that might have impacted the results is that problems given to participants were already formulated in text. This text is already a fairly precise representation of the problem, and therefore our study cannot cast light on earlier stages of the problem comprehension process. More research is needed to understand how people make sense or derive problems from situations.

Another practical aspect of the experimental design to notice when interpreting our results is age and gender balance. Our recruitment strategy was to balance gender and age when possible, but the sign-ups resulted in more females in non-CS than in the CS and CP cohort. Also, the age group in CP was higher than in the other two, which is a natural consequence of constraint programming knowledge being more specific and only encountered at a later stage in education.

Finally, our findings are directly applicable to discrete constraint problem representations only, since we only asked participants to represent constraint problems. Constraint programming is a broad paradigm that covers a large set of problems common in real life (e.g., scheduling, resource allocation, multi-objective optimization), but there are types of problems that are not easily represented through this paradigm (e.g., data-fitting, simulation) and problem kinds that are impractical or impossible to compute through constraint programming, namely, problems in PSPACE or those where the answers sought are unbounded or harder than NP-complete. Nevertheless, we believe that some of the

patterns that we observed for constraint problem specification will also show up when people try to describe other problem categories.

As a next step, we plan to design a visual language for discrete constraints and a system for the input and visualization of such problems, based on the finding from this study. Such a visual language and system will allow for evaluation of the finding and refining of the visual language, as well as exploring any learning effects in participants.

9 Conclusion

In this paper we have analyzed how people create representations of constraint problems. The analysis is made under the assumption that understanding the ways in which unconstrained participants attempt to model problems is useful for designers of visual languages for problem specification. The observations reveal interesting patterns, and provide pointers for the design of future languages for problem specification, be it for facilitating human-to-human communication about problems or to create better UIs that improve and broaden access to constraint modelling to non-specialists.

Some of the main findings are that participant's diagrams are generally not very good problem specifications but show regularities that might be useful for design, that people naturally integrate maths and textual language in their specifications, that people seem to naturally resort to symbols, besides textual labels, in order to provide naming and abstraction of different types of parts of the problem language, and that there is implicit information in gestures, proximity and temporal order that can make problem specifications more complete.

In addition to these findings, we contribute:

- A categorization of visual elements useful for analysis of visual language in this domain.
- A high-level language description of how problems are specified by 3 different expertise groups.
- Entropy measures of the mappings between the two above.
- Suggestions for designers of visual problem specification languages.

Acknowledgments

We thank participants, Steve Linton, Uta Hinrichs, César del Pino, Aaron Quigley and the members of the SACHI research group. This work is supported by EPSRC grants DTG1796157 and EP/P015638/1.

References

[1] J. L. Adams. *Conceptual blockbusting: A guide to better ideas*. WW Norton & Company, 1980.

[2] Ö. Akgün, I. Miguel, C. Jefferson, A. M. Frisch, and B. Hnich. Extensible automated constraint modelling. In *Proceedings of theTwenty-Fifth AAAI Conference on Artificial Intelligence*, pp. 4–11. AAAI Press, 2011.

[3] K. Apt and M. Wallace. *Constraint Logic Programming using ECLiPSe*. Cambridge University Press, 2006.

[4] G. J. Badros, A. Borning, and P. J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, Dec. 2001. doi: 10.1145/504704.504705

[5] D. H. Ballard, M. M. Hayhoe, P. K. Pook, and R. P. Rao. Deictic codes for the embodiment of cognition. *Behavioral and Brain Sciences*, 20(4):723–742, 1997.

[6] A. Bauer, V. Botea, M. Brown, M. Gray, D. Harabor, and J. Slaney. An integrated modelling, debugging, and visualisation environment for g12. In *International Conference on Principles and Practice of Constraint Programming*, pp. 522–536. Springer, 2010.

[7] J. Bertin. *Semiology of Graphics: Diagrams, Networks, Maps*. ESRI Press, 2011.

[8] A. Biere, M. Heule, and H. van Maaren. *Handbook of Satisfiability*, vol. 185. IOS Press, 2009.

[9] A. Borning. Graphically defining new building blocks in thinglab. *SIGCHI Bull.*, 19(1):75–, July 1987.

[10] A. Borning. The programming language aspects of thinglab, a constraint-oriented simulation laboratory. In *Readings in Artificial Intelligence and Databases*, pp. 480–496. Elsevier, 1988.

[11] M. Boshernitsan and M. S. Downes. Visual programming languages: a survey. 2004.

[12] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang, and P. V. Zee. Scaling up visual programming languages. *Computer*, 28(3):45–54, Mar. 1995. doi: 10.1109/2.366157

[13] S. K. Chang, M. M. Barnett, S. Levialdi, K. Marriott, J. J. Pfeiffer, and S. L. Tanimoto. The future of visual languages. In *Proceedings 1999 IEEE Symposium on Visual Languages*, pp. 58–61, Sept. 1999. doi: 10.1109/VL.1999.795875

[14] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko. Let's go to the whiteboard: how and why software developers use drawings. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 557–566. ACM, 2007.

[15] A. Clark. *Supersizing the mind: Embodiment, action, and cognitive extension*. Oxford University Press, USA, 2008.

[16] Cycling '74. Max Software Tools for Media — Cycling '74. `https://cycling74.com/products/max/`, 2018-08-08.

[17] M. G. de la Banda, K. Marriott, R. Rafeh, and M. Wallace. The modelling language zinc. In *International Conference on Principles and Practice of Constraint Programming*, pp. 700–705. Springer, 2006.

[18] F. De Saussure. *Course in general linguistics*. Columbia University Press, 2011.

[19] A. M. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, and I. Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, Sep 2008. doi: 10.1007/s10601-008-9047-y

[20] I. P. Gent, C. Jefferson, and I. Miguel. Minion: A fast, scalable, constraint solver. In *Proceedings of the 2006 conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29–September 1, 2006, Riva del Garda, Italy*, pp. 98–102. IOS Press, 2006.

[21] D. Gentner and A. L. Stevens. *Mental models*. Psychology Press, 1983.

[22] M. L. Gick and K. J. Holyoak. Analogical problem solving. *Cognitive Psychology*, 12(3):306 – 355, 1980. doi: 10.1016/0010-0285(80)90013-4

[23] A. Gleixner, M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. E. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano, J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 6.0. ZIB-Report 18-26, Zuse Institute Berlin, July 2018.

[24] S. Goldin-Meadow, H. Nusbaum, S. D. Kelly, and S. Wagner. Explaining math: Gesturing lightens the load. *Psychological Science*, 12(6):516–522, 2001.

[25] S. Goodwin, C. Mears, T. Dwyer, M. G. de la Banda, G. Tack, and M. Wallace. What do constraint programming users want to see? exploring the role of visualisation in profiling of models and search. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):281–290, Jan 2017. doi: 10.1109/TVCG.2016.2598545

[26] D. Grijincu, M. A. Nacenta, and P. O. Kristensson. User-defined interface gestures: Dataset and analysis. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces*, pp. 25–34. ACM, 2014.

[27] J. Heer, J. Mackinlay, C. Stolte, and M. Agrawala. Graphical Histories for Visualization: Supporting Analysis, Communication, and Evaluation. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1189–1196, Nov. 2008. doi: 10.1109/TVCG.2008.137

[28] F. S. Hillier and G. J. Lieberman. *Introduction to Operations Research*. McGraw-Hill, 9 ed., 2010. International Edition.

[29] D. D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, 3(1):69–101, 1992.

[30] J. Hoffswell, A. Borning, and J. Heer. Setcola: High-level constraints for graph layout. In *Computer Graphics Forum*, vol. 37, pp. 537–548. Wiley Online Library, 2018.

[31] R. E. Horn. Visual language. *MacroVu Inc. Washington*, 1998.

[32] S. Huron, S. Carpendale, A. Thudt, A. Tang, and M. Mauerer. Constructive visualization. In *Proceedings of the 2014 Conference on Designing Interactive Systems*, DIS '14, pp. 433–442. ACM, New York, NY, USA, 2014. doi: 10.1145/2598510.2598566

[33] C. Jefferson, I. Miguel, B. Hnich, T. Walsh, and I. P. Gent. CSPLib: A problem library for constraints. `http://www.csplib.org`, 1999.

[34] D. Kaiser. Physics and Feynman's Diagrams: In the hands of a postwar generation, a tool intended to lead quantum electrodynamics out of a decades-long morass helped transform physics. *American Scientist*, 93(2):156–165, 2005.

[35] G. Kiczales. Beyond the black box: open implementation. *IEEE Software*,

13(1):8–11, Jan 1996. doi: 10.1109/52.476280

[36] D. Kirsh and P. Maglio. On distinguishing epistemic from pragmatic action. *Cognitive Science*, 18(4):513–549, Oct. 1994. doi: 10.1016/0364-0213(94)90007-8

[37] Z. Kiziltan, M. Lippi, and P. Torroni. Constraint Detection in Natural Language Problem Descriptions. In *IJCAI*, pp. 744–750, 2016.

[38] A. Kohnle and G. Passante. Characterizing representational learning: A combined simulation and tutorial on perturbation theory. *Physical Review Physics Education Research*, 13(2):020131, 2017.

[39] R. Kruger, S. Carpendale, S. D. Scott, and S. Greenberg. Roles of Orientation in Tabletop Collaboration: Comprehension, Coordination and Communication. *Computer Supported Cooperative Work (CSCW)*, 13(5-6):501–537, Dec. 2004. doi: 10.1007/s10606-004-5062-8

[40] R. Laban. *The Mastery of Movement*. Dance Books Ltd, Alton, 4th ed. edition ed., Mar. 2011.

[41] G. Lakoff and R. Núñez. *Where Mathematics Comes from: How the Embodied Mind Brings Mathematics Into Being*. Basic Books, 2000.

[42] C. A. L. Lee. The Science and Art of the Diagrams: Culturing Physics and Mathematics, Part 1, 2013.

[43] Z. Liu and J. Stasko. Mental models, visual reasoning and interaction in information visualization: A top-down perspective. *IEEE Transactions on Visualization & Computer Graphics*, (6):999–1008, 2010.

[44] A. Manches and C. O'malley. Tangibles for learning: a representational analysis of physical manipulation. *Personal and Ubiquitous Computing*, 16(4):405–419, 2012.

[45] A. Manches, C. O'Malley, and S. Benford. The role of physical representations in solving number problems: A comparison of young children's use of physical and virtual materials. *Computers & Education*, 54(3):622 – 640, 2010. doi: 10.1016/j.compedu.2009.09.023

[46] K. Marriott and B. Meyer. *Visual language theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[47] S. Martello and P. Toth. Knapsack problems: Algorithms and computer interpretations. *Hoboken, NJ: Wiley-Interscience*, 1990.

[48] Mathworks. Simulink - Simulation and Model-Based Design - MATLAB & Simulink. `https://www.mathworks.com/products/simulink.html`, 2018-08-08.

[49] M. L. McHugh. Interrater reliability: the kappa statistic. *Biochemia Medica*, 22(3):276–282, Oct. 2012.

[50] D. W. McIntyre and M. M. Burnett. Visual Programming. *Computer*, 28(3):14–16, 1995.

[51] G. G. Méndez, U. Hinrichs, and M. A. Nacenta. Bottom-up vs. top-down: Trade-offs in efficiency, understanding, freedom and creativity with infovis tools. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pp. 841–852. ACM, New York, NY, USA, 2017. doi: 10.1145/3025453.3025942

[52] G. G. Méndez, M. A. Nacenta, and U. Hinrichs. Considering agency and data granularity in the design of visualization tools. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pp. 638:1–638:14. ACM, New York, NY, USA, 2018. doi: 10.1145/3173574.3174212

[53] G. G. Méndez, M. A. Nacenta, and S. Vandenheste. ivolver: Interactive visual language for visualization extraction and reconstruction. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pp. 4073–4085. ACM, 2016.

[54] T. Munzner. *Visualization analysis and design*. AK Peters/CRC Press, 2014.

[55] B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1(1):97–123, Mar. 1990. doi: 10.1016/S1045-926X(05)80036-9

[56] B. A. Nardi. *A small matter of programming: perspectives on end user computing*. MIT press, 1993.

[57] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard cp modelling language. In *Principles and Practice of Constraint Programming – CP 2007*, pp. 529–543. Springer Berlin Heidelberg, 2007.

[58] P. Nightingale, O. Akgün, I. P. Gent, C. Jefferson, I. Miguel, and P. Spracklen. Automatically improving constraint models in Savile Row. *Artificial Intelligence*, 251:35–61, 2017.

[59] W. Oechslin. Euclid's geometry - not a via regia? 2010.

[60] V. I. Pavlovic, R. Sharma, and T. S. Huang. Visual interpretation of hand gestures for human-computer interaction: A review. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (7):677–695, 1997.

[61] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[62] R. Penrose. Applications of negative dimensional tensors. *Combinatorial mathematics and its applications*, 1:221–244, 1971.

[63] G. Polya. *Induction and Analogy in Mathematics: Vol. 1 of Mathematics and Plausible Reasoning*. Ishi Press, 1954.

[64] J.-F. Puget. Applications of constraint programming. In *International Conference on Principles and Practice of Constraint Programming*, pp. 647–650. Springer, 1995.

[65] F. K. Quek. Toward a vision-based hand gesture interface. In *Virtual reality software and technology*, pp. 17–31. World Scientific, 1994.

[66] E. D. Ragan, A. Endert, J. Sanyal, and J. Chen. Characterizing Provenance in Visualization and Data Analysis: An Organizational Framework of Provenance Types and Purposes. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):31–40, Jan. 2016. doi: 10.1109/TVCG.2015.2467551

[67] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for all. *Commun. ACM*, 52(11):60–67, Nov. 2009. doi: 10.1145/1592761.1592779

[68] S. I. Robertson. *Problem solving: perspectives from cognition and neuroscience*. Psychology Press, 2016.

[69] F. Rossi, P. van Beek, and T. Walsh, eds. *Handbook of Constraint Programming*. Elsevier, 2006.

[70] D. J. Rough and A. J. Quigley. Jeeves-an experience sampling study creation tool. *BCS Health Informatics Scotland (HIS)*, 2017.

[71] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.

[72] M. Shishmarev, C. Mears, G. Tack, and M. Garcia de la Banda. Visual search tree profiling. *Constraints*, 21(1):77–94, Jan 2016.

[73] K. Stenning and J. Oberlander. A Cognitive Theory of Graphical and Linguistic Reasoning: Logic and Implementation. *Cognitive Science*, 19(1):97–140. doi: 10.1207/s15516709cog1901_3

[74] I. E. Sutherland. Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE design automation workshop*, pp. 6–329. ACM, 1964.

[75] B. Tversky. What do sketches say about thinking. In *2002 AAAI Spring Symposium, Sketch Understanding Workshop, Stanford University, AAAI Technical Report SS-02-08*, pp. 148–151, 2002.

[76] J. S. Uebersax. Diversity of decision-making models and the measurement of interrater agreement. *Psychol Bull*, 101(1):140–146, 1987.

[77] J. S. Uebersax. Kappa Coefficients: A Critical Appraisal. `http://www.john-uebersax.com/stat/kappa.htm`, 2018-08-03.

[78] VERBI Software. MAXQDA 2018. `https://www.maxqda.com`, 2017.

[79] M. Wallace. Practical applications of constraint programming. *Constraints*, 1(1-2):139–168, 1996.

[80] J. Walny, S. Carpendale, N. H. Riche, G. Venolia, and P. Fawcett. Visual thinking in action: Visualizations as used on whiteboards. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2508–2517, 2011.

[81] J. Walny, J. Haber, M. Dörk, J. Sillito, and S. Carpendale. Follow that sketch: Lifecycles of diagrams and sketches in software development. In *Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on*, pp. 1–8. IEEE, 2011.

[82] J. Walny, S. Huron, and S. Carpendale. An exploratory study of data sketching for visual representation. In *Computer Graphics Forum*, vol. 34, pp. 231–240. Wiley Online Library, 2015.

[83] Y. Wang, Y. Wang, Y. Sun, L. Zhu, K. Lu, C.-W. Fu, M. Sedlmair, O. Deussen, and B. Chen. Revisiting stress majorization as a unified framework for interactive constrained graph visualization. *IEEE transactions on visualization and computer graphics*, 24(1):489–499, 2018.

[84] C. Ware. *Information Visualization: Perception for Design*. Elsevier, May 2012. Google-Books-ID: UpYCSS6snnAC.

[85] J. M. Wolfe. Guided Search 2.0 A revised model of visual search. *Psychonomic Bulletin & Review*, 1(2):202–238, June 1994. doi: 10.3758/BF03200774

[86] S. Wolfram. Mathematical notation: Past and future. In *MathML and Math on the Web: MathML International Conference, Urbana Champaign, USA*, 2000.

[87] J. Zhang and D. A. Norman. Representations in distributed cognitive tasks. *Cognitive science*, 18(1):87–122, 1994.