Symbolism for modelling, reformulations, and parallelism: MaxiCP-Modelling

Guillaume Derval
Montefiore Research Unit, ULiège, Belgium

Damien Ernst
Montefiore Research Unit, ULiège, Belgium

Abstract

We present our ongoing work on MaxiCP-Modelling, a symbolic modelling layer for the MaxiCP solver. Models in MaxiCP-Modelling consist of a functional linked list of symbolic constraints; the models can later be concretised on an underlying solver. Decisions made during the resolution process can be progressively added to these lists to create new models that represent the current status of the solver. This mechanism creates a functional tree of models. We show how to take advantage of this framework in the context of parallel and distributed resolution, pre-processing, and the use of multiple types of solvers (hybridisation).

2012 ACM Subject Classification Mathematics of computing → Solvers; Computing methodologies → Symbolic and algebraic manipulation

Keywords and phrases Symbolic modelling, reformulation, parallelism, distribution, immutability

Acknowledgements The authors want to thank Pierre Schaus and Augustin Delecluse who provided feedback during development.

1 Introduction

This paper presents ongoing work on MaxiCP-Modelling, a modelling layer built above MaxiCP[1].

MaxiCP-Modelling emphasises the importance of making *models* (symbolic representations of problems) a priority - the so-called first-class citizens, and to ensure these models are immutable and serializable, so they can be easily shared between multiple threads, processes, or machines. The concepts underlying MaxiCP-Modelling originated from a Constraint Programming research project that involves hybridisation, automated and real-time reformulation of problems, and parallel computing. To continue this line of research effectively, a modelling language that simplifies and seamlessly integrates these functionalities is crucial. Specifically, we need:

- To treat problems/models as first-class citizens: This allows for straightforward reformulation and the development of algorithms that can deduce characteristics about the problem structures.
- To implement this in a host language, not an ad hoc one: the chosen language needs to be commonly used to maximise the reach and impact of our work.
- To use the same model (and variations of it) multiple times concurrently, in parallel or in a distributed manner.
- To enable the application of various types of solvers to the same models.
- To enable interaction between different solvers via the modelling layer.

[1] MaxiCP is an in-development solver, derived from the MiniCP[7]. While MiniCP was made for education-related purposes, MaxiCP aims at being its research-focused counterpart.

- To maintain deep interaction with underlying solvers, particularly with Constraint Programming (CP) ones: this ensures that we can continue to leverage the full potential of these problem-solving tools. A particular point of attention is that we want to be able to be able to interact at the inference/fixed-point-algorithm level.

The main contribution of this paper is the formalisation of the concept of model (approximately linked lists of symbolic constraints) in MaxiCP-Modelling, and the tools used to interact with them.

Related works & comparison with other languages and frameworks

The Constraint Programming community has indeed extensively researched the subject of modelling layers, frameworks, languages, and techniques throughout its history. This has resulted in a considerable number of tools, each with their own strengths and weaknesses. Most solvers, such as Choco[12], Gecode[5], and OscaR[10], offer well-designed domain-specific languages. However, these are usually focused on the solving process and typically do not provide the ability to directly manipulate the *structure* of a problem. Moreover, they rarely offer hybridisation capabilities.

MiniZinc[9] is an ad hoc, independent modelling language that is not tied to a specific underlying solver. Compatible solvers can ingest a flattened version of MiniZinc, called FlatZinc, that is generated by the MiniZinc compiler. As an independent domain-specific language, rather than a usual programming language, MiniZinc thus has limited potential for in-depth interactions with solvers. Moreover, when MiniZinc models are translated into their flattened counterpart, FlatZinc, a significant portion of the original structure is lost. This poses a dilemma: we are left with either a structured but complex-to-parse language or a simpler, easy-to-parse language devoid of useful structural information. XCSP3[1] is another attempt at an independent language. XCSP3 is declarative rather than imperative: it structures models/problems in XML files that explicitly focus on structures. It has some capabilities to indicate search heuristics, like MiniZinc, but it is similarly limited when interacting directly, at runtime, with solvers. It is, however, far easier to parse in its structured form. Some works have already used tools like MiniZinc/FlatZinc and XCSP3 to work on the structures of problems at runtime. As an example, see the first implementation of Embarrassingly Parallel Search[14] that takes a FlatZinc file and divides it in multiple FlatZinc files to be solved independently. Even with XCSP3, this kind of solution mostly forbids interacting deeply with solvers and requires multiple compilation/parsing passes, which are sometimes wasteful and cumbersome. MiniZinc and XCSP3 thus do not fulfil our requirements; it is, however, important to note that they accomplish one feat that is probably incompatible with our requirements: they are able to work with many, many kinds of solvers and provide a way to build benchmarks for them.

MaxiCP-Modelling is related to three more recent attempts at creating new modelling and solving languages. The first is CPMPy[6], a Python library that checks almost all the requirements listed above: models are a first-class citizen, can be instantiated into various solvers, that can be used concurrently and interacted with. The amount of interactivity allowed is, however, limited by the agnostic stance of the library; it does not assume anything about the underlying solver, and has no concept of search trees or even search heuristics. These can still be defined by directly accessing the underlying solver (which is allowed by the library and by the use of a "true" programming language like Python), but this access is itself limited by the API exposed by the solvers. Currently CPMPy supports OR-Tools [11] as its main CP solver; OR-Tools is in C++, not in Python, so the amount of interactivity is constrained by the existing APIs. This, of course, can be circumvented, but at the price of extending OR-Tools in addition to CPMPy.

In 2013, Michel & Van Hentenryck introduced Objective-CP[15], a new CP library built around the Objective-C language. MaxiCP-Modelling takes many ideas from Objective-CP such as *concretizations* (which can actually be traced back to Comet[8]). Objective-CP focuses a lot on the way search strategies are defined, which is not an important point in our context. An extension to Objective-CP, Ocpmcl[4] introduces the concept of Model Combinators, that allows for the linking together of different solvers running concurrently in a declarative way. We have yet to implement this in MaxiCP-Modelling (mainly because it is too early in the development and because it does exclude some useful scenarios like hybridisation at the fixed-point level) but it is definitely a goal.

A third source of inspiration is OscaR-Modelling (whose architecture was never formally published), the modelling layer of the OscaR solver, from which MaxiCP-Modelling borrows a lot of implementation ideas.

2 Theoretical Framework

In this section, we formalise the concept of *model* in MaxiCP-Modelling as a constraint satisfaction/optimisation problem that sightly differs from the usual definition, but is as expressive. Let us first define the concepts of variables and constraints:

▶ **Definition 1** (Variable). *A (integer) variable v is an object with a domain $D_v \in \mathbb{Z}$, which is the set of possible values that v can represent.*

▶ **Definition 2** (Constraint). *A constraint $c = \langle V_c, R_c \rangle$, defined over the subset of variables $V_c = \{v_1, v_2, \ldots, v_k\}$, is represented by a relation $R_c \subseteq D_1 \times D_2 \times \ldots \times D_k$, the set of domain value selections that satisfy the constraint.*

We use a recursive definition of what a model is. Intuitively, a model *inherits* from another model and its constraints, by adding a new constraint to an existing model:

▶ **Definition 3** (Models). *A satisfaction model M is a tuple $\langle c_M, parent(M) \rangle$ containing a constraint and another satisfaction model, called the parent model. The root model $M_\emptyset$ is defined as an empty model, without constraints or parent. A model thus implicitly represents a list of constraints, accessible via $ctrs(M) = \{c_M\} \cup ctrs(parent(M))$, with $ctrs(M_\emptyset) = \emptyset$.*

We say that M is the child of parent(M). All the children of a model, and their own children, ..., recursively, are called the offspring of the model. The parent, grandparent, ..., models are called ancestors. Model loops are prohibited: a model cannot be an offspring/ancestor of itself.

The function $vars(M) = \bigcup_{c \in cstrs(M)} V_c$ returns the list of variables in M. An assignment of a model A is a tuple assigning a value to each variable, A_i being the value assigned to variable $v_i \in vars(M)$. An assignment A is a solution if it respects all the constraints in the model, i.e. $\forall c \in ctrs(M) : \{A_i \mid v_i \in V_c\} \in R_c$. The set of all solutions is given by $sols(M)$.

An optimisation model is a tuple containing a satisfaction model M and an objective function $f_M : \mathbb{Z}^{|vars(M)|} \to \mathbb{R}$, which provides a total order on $sols(M)$. Optimal solutions are defined as those being the smallest in the set.

For simplicity, in the remainder of this paper, we focus in the remainder of this paper on satisfaction problems; the definitions and reasoning can be adapted for optimisation problems at the expense of some verbosity, which we wish to avoid here.

This model definition is purely *symbolic* (it is purely represented by its variable with their default values and a list of constraints) and is decoupled from any resolution technique.

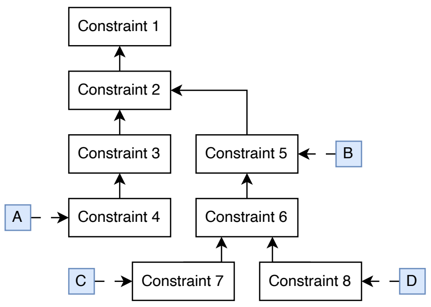

Figure 1 An example of a model tree. A model in MaxiCP-Modelling is a tuple composed of a constraint and a link to another model. Each white node in this example is thus a model. Some of them have been named A,B,C and D. The model A is thus composed of constraints 1-4, the model B of constraints 1-2 and 5, the model C of constraints 1-2 and 5-7, and the model D of constraints 1-2, 5-6 and 8.

The model implicitly defines its solutions but does not describe how to compute them: it is purely declarative. Models form a model tree rooted on the model $M_\emptyset$, as shown in the example in Figure 1.

2.1 Transformations & Concretisations

We define two additional types of operations on models: transformations, which allow for the preprocessing of a model and modification of its content, and concretisations, which instantiate the model in a solver and allow it to be solved.

▶ **Definition 4** (Transformation). *A transformation $t(M, \cdot) = M'$ produces a model M' based on an existing model M. The new model may or may not share an ancestor with the old one, except for the root model $M_\emptyset$ (by definition of a model). A transformation is termed relaxing if the set of solutions is expanded after the transformation ($sols(M) \subseteq sols(M')$), equivalent if the set is identical ($sols(M) = sols(M')$), and restrictive if the set is reduced ($sols(M') \subseteq sols(M)$). Any transformation must keep the empty model as-is: $t(M_\emptyset, \cdot) = M_\emptyset$.*

An online transformation is a transformation $ot(M_1, M'_1, M_2, \cdot) = M'_2$ that, after having converted an initial model M_1 to another model M'_1, can take another model M_2 that is a child of M_1 and transform it into a model M'_2 that is itself a child of M_2. It can be used as a standard transformation by setting $t(M, \cdot) = ot(M_\emptyset, M_\emptyset, M, \cdot)$.

In general, transformations are not required to be stateless, particularly online ones, i.e. they may require to maintain additional information to work properly between calls. Moreover, some of them may have to introduce, remove, or modify the meaning of variables; in which case, they provide a method to recover the value of the initial symbolic variables from the transformed model and its children.

▶ **Definition 5** (Concretization). *A concretization $\mathcal{M}$ of a model M is an instantiation of the model in a given solver (that can be a constraint programming solver, an integer linear programming solver, or any other type of solver). In a concretisation, variables are mutable and can be updated by the solver; constraints are parts of the solver algorithms that modify the domain of the variables.*

In order to differentiate the variables/constraints "outside" and "inside" the solver, we call them *symbolic* or *concrete* variables/constraints respectively.

It is not expected that every type of solver will be capable of concretising every model. For instance, a linear programming solver is not capable of concretising non-linear constraints. However, an ad hoc (equivalent or relaxing) transformation may be applied to make a model compatible, for instance, by linearising or removing certain constraints.

A concretisation is a stateful (i.e. an object maintaining state between interactions), mutable object: it is possible to interact with it. The main operations of any concretisation are:

- Retrieving information about the domains of the (concrete) variables in the underlying solver, which are in general more precise than the ones available in the related symbolic variables
- Adding a new constraint (or, equivalently, jumping to a *child model*, that is adding all the constraint between the currently concretised model and another symbolic one that inherits from the first)
- Saving the current state (that is, storing the list of currently concretised constraints) and being able to roll back to previously saved states, in a stack-like manner (as is common in CP solvers).

Each concretisation may have particular methods attached to it, related to the underlying solver.

2.2 Partially Symbolic Concretisations

Certain categories of solvers, such as typical constraint programming or integer linear programming solvers, fundamentally employ a branch-and-bound algorithm (or a variant). They have a method that infers properties about the variables and constraints from the current model; this method helps reduce the variable domains without excluding any solutions. At the end of this inference phase, it is rare that the problem is solved, i.e., is it rare that the Cartesian product of the domains forms the set of solutions. The branch-and-bound algorithm then performs a *branching operation*: it operates a modification on the model (typically adding a constraint), then calls the inference algorithm recursively, and when that call is completed, it performs the opposite operation (generally adding the negation of the constraint) and again calls itself recursively. Using the definitions above, this branching behaviour of model M can be seen as the creation of two models $M_{\text{left}} = \langle c, M \rangle$ and $M_{\text{right}} = \langle \neg c, M \rangle$, both children of M. The search tree created by the branch-and-bound algorithm thus forms a tree of models as defined above.

We call the operation of instantiating a model in a solver working in this way (or in a similar way, for instance for an algorithm reordering the sequence of separated problems) a *partially symbolic concretisation*. Such a concretisation of a model M allows interaction during the resolution: at any given moment, the underlying solver has a current model M_c, a set of *open* models yet unexplored $\mathcal{M}_{\text{waiting}}$, and a set of discovered solutions $\mathcal{S}$, so that $\text{sols}(M) = \text{sols}(M_c) \cup \mathcal{S} \cup (\bigcup_{M_a \in \mathcal{M}_{\text{waiting}}} \text{sols}(M_a))$.

While a Partially Symbolic Concretisation is still a concretisation, that is, a mutable object, the various elements introduced here are purely symbolic and immutable. At any time, the mutable state of the underlying solver is represented by an immutable model. The state of the solver can only be changed by creating a new model, inheriting from the previous one, or by directly creating a new lineage of a model from an ancestor or from $M_\emptyset$ (transformation). The immutable aspect is useful in a context of parallelisation/distribution [13], as it helps avoid race conditions, among other things. Furthermore, *partially symbolic concretisations* will allow us to interact with the solver and simply divide the problems during runtime while continuing to benefit from these aspects of immutability. Moreover, it paves the way to easy hybridisation, as we explain in the following subsection.

2.3 Proxies

Proxies are stateful, mutable objects that have the same interface as concretisations. Their role is to redirect and (optionally) modify methods to other concretisations or proxies. We separate the proxies into two kinds:

- First are *transformer proxies*, whose role is to use an underlying (online) transformer and to redirect the newly created model (or model lineage) to another concretisation or proxy. They take care of transparently handling changes to the meaning of variables due to the transformation, and transfer state-saving or rollback requests.
- Second are *one-to-n proxies*, that redirect new constraints and state-saving or rollback requests to multiple other proxies or concretisations. They also unify the various domains being retrieved for each variable (by taking their intersection).

Proxies enable us to perform *hybridisation*: using various kinds of solvers at the same time, and making them interact at will through the modelling layer. We could use a CP concretisation of a model, then transform this main model into an LP relaxation (via a transformer proxy and an LP concretisation of the relaxed model), and use this relaxation as an external constraint on the objective, by linking the two with a one-to-two proxy and a little amount of code.

Proxies are a lower-level concept than Model Combinators[4]: they are more complex to use, but are less constrained (notably they can work at the search-tree and fixed-point level if needed), and provide a suitable base to implement Model Combinators in a proper way later in the development process.

2.4 In summary

Various kinds of links and interactions between solvers can be created in this framework: hybridisation with other solvers, with multiple relaxations, online hybridisation (online) autotabling, etc. Moreover, the immutable symbolic models allow for easy parallelism and distribution: they remove the need for most synchronisation mechanisms and can be made easy to serialise. Software using this framework are (or, at least, can be) able to reason upon constraints in isolation of a solver, and to modify models before, during and after their concretisation for underlying solvers.

3 Implementation

We rapidly describe here how the various aspects of immutable models as defined above are implemented in MaxiCP-Modelling. We use the underlying solver MaxiCP, based on MiniCP [7], made in Java. This solver has no notion of symbolism; when a constraint is created, it registers itself (via callbacks) on the variables that are in its scope, and then the solver directly loses the pointer to the constraint: the solver is not able, for example, to enumerate the currently active constraints or to know their scope. Each variable has a list of callbacks to perform that does not allow other interactions. The variables have a mutable state: the solver is intended to be used by a single thread of execution at a time. In MaxiCP, the notion of model is mixed with the notion of a solver instance. These comments apply to MaxiCP as well as other solvers like Choco[12] or Gecode[5].

Clearly, these definitions of variables/constraints/models are incompatible with those defined at the beginning of this document. We will call the variables/constraints of the underlying solver (here MaxiCP) *concrete* variables/constraints, which we will differentiate from *symbolic* variables/constraints, which we will implement on top, in MaxiCP-Modelling.

This additional layer, called the *modelling* layer, is designed to preserve the properties of MaxiCP: accessibility to various parts of the underlying solver, short distance between abstraction and implementation, execution speed. The modelling layer does not restrict any existing functionality, but adds new ones.

When using MaxiCP-Modelling, the user must first create a *context*, which is effectively a pointer to the current model. The current model is the one reflected in various interactions with MaxiCP-Modelling: inspection of variables, enumeration of constraints... As models are immutable, adding a constraint means creating a new model with this constraint, inheriting from the current model, and making the *context* point to the newly created model.

Symbolic variables in MaxiCP-Modelling are implemented as *relays* with a *default domain*. A model, as defined in the theoretical framework above, does not define (directly) a domain for variables, except implicitly through constraints. However, for speed and simplicity, it is allowed in MaxiCP-Modelling to define a default domain that will serve to instantiate the concrete domains of the underlying solver. Symbolic variables are immutable: only introspection functions, such as `hasValue(v)` or `getMin()`, are available. The behaviour of these methods depends on the *context*: if it points to a symbolic model, method calls on the variable will work on the default domain; if it points to a concrete model, calls are redirected to the underlying concrete variable.

The *context* is in practice a *thread-local* variable: each thread has its own current model, which may/will differ from other threads. This mechanism, in addition to variables that redirect calls to concrete variables, allows for the implementation of parallelism using a single instance of symbolic variables: everything being perfectly immutable, there is no competition problem, and all threads can use the same instance. For concretised models, only one thread is allowed to have them as context; but the user still uses the symbolic variable the entry point to the solver. Listing 1 shows an example of the API currently proposed to the end user.

Listing 1 Usage of MaxiCP-Modelling: implementation of a n-queens model

```
int n = 12; Context context = Factory.makeContext();

IntVar[] q = context.intVarArray(n, n);
IntExpression[] qLeftDiagonal = IntStream.range(0, q.length)
    .mapToObj(i -> q[i].plus(i)).toArray(IntExpression[]::new);
IntExpression[] qRightDiagonal =  IntStream.range(0, q.length)
    .mapToObj(i -> q[i].minus(i)).toArray(IntExpression[]::new);

//Here, the context is symbolic. Adding new constraints actually
//creates new symbolic models, inheriting from each other
context.add(new AllDifferent(q));
context.add(new AllDifferent(qLeftDiagonal));
context.add(new AllDifferent(qRightDiagonal));

//Custom branching procedure
Supplier<Procedure[]> branching = () -> {
  int idx = -1; // index of the first variable that is not fixed
  for (int k = 0; k < q.length && idx == -1; k++)
    if (!q[k].isFixed())
```

```
        idx=k;
    if (idx == -1) return EMPTY;
    else {
      //Even if this branching procedure is used in a concretised
      //context, we can work with the (symbolic) IntVar and
      //IntExpression that will transparently redirect calls to
      //their methods to the concrete underlying variables
      IntExpression qi = q[idx];
      int v = qi.min();
      Procedure left = () -> context.add(new Eq(qi, v));
      Procedure right = () -> context.add(new NotEq(qi, v));
      return branch(left,right);
    }
};

//Run the CP solver. Multiple threads can do this concurrently
context.runAsConcrete(CPModelInstantiator.withTrailing, (cp) -> {
      //here, the context points to the CP concretised model
      SearchStatistics results = cp.dfSearch(branching).solve();
      System.out.println("Total number of solutions: " +
          results.numberOfSolutions());
});
//here, the contexts again point to the latest symbolic model.
```

4 Status & future work

The work on MaxiCP-Modelling is ongoing, as is the work on the underlying solver MaxiCP. The code for both is non-public: MaxiCP is based on the MiniCP solver made for education, particularly on its teacher version. It thus contains all solutions to the various exercises of the related MOOC. Until it has diverged sufficiently, the code will not be displayed publicly but is still freely available upon request.

The ultimate goal for MaxiCP-Modelling is to be the base tool for a research project about online reformulation, hybridisation and parallelism. The next step mainly consists of improving the API and the "language" (the DSL) of the modelling layer, and to document it thoroughly. We then want to add the concepts of Model Combinators [4] that will greatly simplify the usage.

We have working implementations of Embarrassingly Parallel Search[14, 3] and Work-Stealing on MaxiCP-Modelling, showing that the concepts work in a real-world context. However, we need to streamline their usage, both for the sake of the users and for our own too. Currently, it is only possible to concretise a model into the underlying CP solver, but the implementation of the machinery needed to implement a relaxing transformation and the related concretisation to a linear programming solver is ongoing. We will then tackle Mixed-Integer Linear Programming and Cost Function Networks[2].

References

1 Frederic Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. Xcsp3: An integrated format for benchmarking combinatorial constrained problems, 2022. `arXiv:1611.03398`.

2 Céline Brouard, Simon de Givry, and Thomas Schiex. Pushing Data into CP Models Using Graphical Model Learning and Solving. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 811–827, Cham, 2020. Springer International Publishing. `doi:10.1007/978-3-030-58475-7_47`.

3 Guillaume Derval, Pierre Schaus, and Jean-Charles Régin. Embarassingly Parallel Search Reengineered. In *Doctoral Program of the 22nd International Conference on Principles and Practice of Constraint Programming (CP 2016)*, 2016.

4 Daniel Fontaine, Laurent Michel, and Pascal Van Hentenryck. Model Combinators for Hybrid Optimization. pages 299–314, September 2013. `doi:10.1007/978-3-642-40627-0_25`.

5 Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.

6 Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

7 L. Michel, P. Schaus, and P. Van Hentenryck. Minicp: a lightweight solver for constraint programming. *Mathematical Programming Computation*, 13(1):133–184, 2021. `doi:10.1007/s12532-020-00190-7`.

8 Laurent Michel and Pascal Van Hentenryck. The Comet Programming Language and System. volume 3709, pages 881–881, October 2005. `doi:10.1007/11564751_119`.

9 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

10 OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

11 Laurent Perron and Vincent Furnon. Or-tools. URL: `https://developers.google.com/optimization/`.

12 Charles Prud'homme and Jean-Guillaume Fages. Choco-solver: A java library for constraint programming. *Journal of Open Source Software*, 7(78):4708, 2022. `doi:10.21105/joss.04708`.

13 Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, February 2004. Google-Books-ID: _bmyEnUnfTsC.

14 Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly Parallel Search. In Christian Schulte, editor, *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 596–610, Berlin, Heidelberg, 2013. Springer. `doi:10.1007/978-3-642-40627-0_45`.

15 Pascal Van Hentenryck and Laurent Michel. The Objective-CP Optimization System. pages 8–29, September 2013. `doi:10.1007/978-3-642-40627-0_5`.