

The PaLM system: explanation-based constraint programming

Narendra Jussien and Vincent Barichard

École des Mines de Nantes, 4 rue Alfred Kastler, BP 20722

F-44307 Nantes Cedex 3, France

Narendra.Jussien@emn.fr

WWW home page: <http://www.emn.fr/jussien>

Abstract. Explanation-based constraint programming is a new way of solving constraint problems: it allows to propagate constraints of the problem, learning from failure and from the solver (thanks to recording explanations) and finally allows to get rid of backtrack-based complete searches by allowing more free moves in the search space (while remaining complete). This paper presents the PaLM system, an implementation of an explanation-based constraint programming system in `choco` [16], a constraint programming layer on top of `claire` [3].

1 Introduction to PaLM programming

The easiest way to present the PaLM (Propagate and Learn with Move) system is to show its main three features: it is a *constraint programming* system, able to *explain its behavior* and able to *handle dynamic constraint additions and removals* (following code excerpts are written in `claire` [3]).

The first feature of the PaLM system is that it can be considered as a **constraint programming system**. Let consider a very simple scheduling problem: it consists in tasks a , b , c , d , e and f to be scheduled following the precedence constraints states in Table 1. Task f can be considered as the final completion date of the project. The problem is to determine the earliest ending time of the project. This problem can be modeled as a constraints system. In the PaLM system, it is modeled as presented in Figure 1.

Task	Duration	Ancestors	Task	Duration	Ancestors
a	1	-	d	4	$a b$
b	2	-	e	3	c
c	1	-	f	-	$d e$

Table 1. Data for a simple scheduling problem

```

[schedulingProblem() ->
  let pb := makePalmProblem("scheduling", 6),
      a  := makePalmIntVar(pb, "a", 1, 15), // bounds of the variables
      ...
      f  := makePalmIntVar(pb, "f", 1, 15)
  in (
    post(pb, d >= a + 1), // stating the precedence constraints
    post(pb, d >= b + 2),
    post(pb, e >= c + 1),
    post(pb, f >= d + 4),
    post(pb, f >= e + 3),

    propagate(pb), // propagate the constraints
    f // the result is variable f
  )]

```

Fig. 1. Coding a simple scheduling problem with PaLM

That code is very similar to the one that would be written in any classical constraint programming language: actually, removing **PaLM** in every term it appears in would give a valid **choco** code. Moreover, when interpreting that code, the **PaLM** system gives the answer shown in figure 2. The **PaLM** system as would have done any other constraint system informs us that our project cannot end before the date 7.

```

-- CLAIRE run-time library v 2.5.1 [os: ntw, C++:MSVC] --
-- CLAIRE interpreter - Copyright (C) 1994-97 Y. Caseau (see about())
Choco version 0.23, Copyright (C) 1999-2000 F. Laburthe
Palm version 0.2.10, Copyright (C) 2000 N. Jussien
palm> schedulingProblem()
eval[1]> f:[7..15] // our project cannot end before date 7
palm>

```

Fig. 2. Answer to a simple scheduling problem

The **PaLM** system provides much more than a simple propagation mechanism: it is able to provide **explanations for basic events**. Suppose that we want to end the project at date 6. In a classical system, the only way to achieve that is to study the constraint system and try to determine why the minimal value of variable f is 7 (that would be easy in our toy problem but not for a real life problem). The **PaLM** system provides tools to help the user in that task. The key concept is the notion of *explanation* as presented in [13]: an explanation E for an information I (current lower bound, current upper bound, value removal, ...) is a set of constraints such that its associated information remains valid as long

as all the constraints in E are effectively active in the constraints system. For example, we can get an explanation of the lower bound of variable f , thanks to the PaLM method `self_explain` (see figure 3).

```
palm> let e := PalmSet() in ( self_explain(f, INF, e), e)
eval[2]> {d >= a + 1, d >= b + 2, f >= d + 4}
palm>
```

Fig. 3. An example of the `self_explain` function

`self_explain` modifies its third parameter (a `PalmSet`) which contains constraints in order to add the explanation of the asked event. The result obtained here shows that the current value of f does not depend on c nor e . The PaLM system is able to explain the current lower bound of a variable, its current upper bound or specific value removals.

From the computed explanation, it seems that in order to reduce the lower bound of f , one should modify the constraints relating a , b , d and f . For example, to decrease the duration of task d , the easiest way is to remove the constraint ($f \geq d + 4$) and replacing it by a new one. This can be done in PaLM using the methods `remove` and `post` that both work incrementally: we can **add or remove constraints** from the constraint system any time before/during/after the resolution. Let `ct` be the constraint $f \geq d + 4$. Figure 4 shows a sequence of removal/addition of constraints and its result on our scheduling problem.

```
palm> remove(ct)
eval[3]> nil // the removal has been dynamically done
palm> f
eval[4]> f: [5..15]
palm> let e := PalmSet() in (self_explain(f, INF, e), e)
eval[5]> {e >= c + 1, f >= e + 3}
palm> (post(pb, f >= d + 2), propagate(pb))
eval[6]> nil
palm> f
eval[7]> f: [5..15]
```

Fig. 4. Removing and adding constraints in our scheduling problem

When adding the new constraint ($f \geq d + 2$), the value of f is not modified, f now depends on c and e as shown in the explanation computed after removing the offending constraint.

The three presented features (constraint propagation, learning by explanations and dynamic adding and removing of constraints) allow us to go further.

For example, when a problem has no solution it can provide a constraint relaxation system by computing an explanation for the empty domain of the failing variable¹ and selecting from that explanation a constraint to be removed according to a given criterion (taking into account weights on constraints, ...). The **PaLM** system provides constraint weight management tools and automatic constraint relaxation mechanisms.

The paper is organized as follows: in Section 2, some theoretical background is recalled for explanations and insight of adding explanation possibilities in a constraint system are given², Section 3 shows that explanations can be used to provide dynamic constraint removal and section 4 tells more on how programming with explanations and without any real backtrack. Next, Section 5 describes the **PaLM** system and its derivatives: **PaLMenum** and **PaLMschedule**.

2 Explanations for constraint systems

The key feature of the **PaLM** system is the notion of **eliminating explanation**. In the following, we consider a CSP (V, D, C) . In our system, choice constraints (eg. value assignments - $v_i = a_i$) are considered as constraints added to the system.

2.1 Nogoods and eliminating explanations

A **nogood** or **contradiction explanation** is a subset of the current constraint system of the problem that left alone leads to a contradiction (no feasible solution can contain a nogood). A nogood can be splitted into two parts: a subset of the original set of constraints ($C' \subset C$ in equation 1) and a subset of choice constraints introduced so far in the search.

$$C \vdash \neg (C' \wedge v_1 = a_1 \wedge \dots \wedge v_k = a_k) \quad (1)$$

For most nogoods³ (the ones that do contain at least one choice constraint) a variable v_j can be selected and the previous formula rewritten as:

$$C \vdash C' \wedge \bigwedge_{i \in [1..k] \setminus j} (v_i = a_i) \rightarrow v_j \neq a_j \quad (2)$$

The left hand side of the implication constitutes an **eliminating explanation** for the removal of value a_j from the domain of variable v_j and is noted $\mathbf{expl}(v_j \neq a_j)$.

¹ Inconsistency of constraint problems is detected when the solver wipes out the domain of a variable.

² Our support language will be **choco** a constraint programming layer on top of **claire**.

³ Note that if a nogood does not contain any choice constraints it means that the current problem does not have any possible solution.

When the domain of variable v_j becomes empty during filtering, a new no-good is deduced from the eliminating explanations of all its removed values:

$$C \vdash \neg \left(\bigwedge_{a \in d(v_j)} \text{expl}(v_j \neq a) \right) \quad (3)$$

There generally exist several eliminating explanations for a given value removal. One may want to record all of them but this leads to an exponential space complexity. Another way relies in *forgetting* (erasing) nogoods that are no longer relevant⁴ to the current variable assignment. By doing so, the space complexity remains polynomial. We therefore keep only **one** explanation at a time for a value removal. In the worst case, the space required to manage nogoods is $O(n^2d)$ where n is the number of variables and d the maximum size of the domains in the CSP. Indeed, the size of each eliminating explanation is at most $(n - 1)$ and there are at most $n \times d$ eliminating explanations: one for each value of each domain.

2.2 Integrating explanation mechanisms

The most interesting eliminating explanations are those who are minimal for the inclusion in order to precisely focus on constraints that really impact the values of a variable. Unfortunately, computing such precise explanations is very time consuming. A good compromise between preciseness and easy computation is to try to use the knowledge embedded in the constraint solver to provide explanations. Indeed, constraint solvers exactly (but not always explicitly) know why they do remove values from domains of considered variables. By explicitly stating that knowledge, very interesting explanations can be computed.

Storing the explanations The first step for integrating explanations within a constraint system like **choco** is to define a new variable class inherited from the one in **choco** that will allow us storing explanations (see Fig. 5).

Computing the explanations The second step consists in specializing the behavior of all the constraints in **choco** in order to add the explanation computation code. Let see those modifications on an example.

Figure 6 shows the **choco** code for awaking a constraint $X \geq Y + c$ after decreasing the upper bound of its first variable (X). If the decreasing of an upper bound is done on the first variable of the constraint (when `idx = 1`) then the upper bound of the second variable (`c.v2`) should be updated too (`c.idx2` gives the index of constraint `c` in variable `c.v2`). The corresponding **PaLM** code is given on figure 7.

⁴ A nogood is said to be relevant if all the assignments in it are still valid in the current search state [1].

```

// The original choco variable
IntVar <: AbstractVar(
  inf:integer,
  sup:integer,
  value:integer = unknown
)
// The inherited PaLM variable
PalmIntVar <: IntVar(
  explanationOnInf:list[PalmExplanation] = nil, // storing structure
  explanationOnSup:list[PalmExplanation] = nil
)

```

Fig. 5. A new variable for PaLM

```

[awakeOnSup(c:GreaterOrEqualxyc, idx:integer) : void
-> if (idx = 1) updateSup(c.v2, c.v1.sup - c.cste, c.idx2)]

```

Fig. 6. The choco code for handling $X \geq Y + c$ when decreasing $X.sup$

The only modification is in the adding of a fourth parameter to the `updateSup` function allowing the association of an explanation to a solver action (decreasing the superior value of a variable). The explanation traces the solver behavior. Indeed, the updating of the superior value of the second variable of the constraint is due to the applied constraint (comment (a) in figure 7) and to the fact that the superior value of the first variable of the constraint has been modified earlier (comment (b) in figure 7).

```

[awakeOnSup(c:PalmGreaterOrEqualxyc, idx:integer) : void
->   if (idx = 1)
      let e := PalmSet() in (
        e :add c,                // (a)
        self_explain(c.v1,SUP,e), // (b)
        updateSup(c.v2, c.v1.sup - c.cste, c.idx2, e)
      )]

```

Fig. 7. The PaLM code for handling $X \geq Y + c$ when decreasing $X.sup$

Once all the propagating code has been augmented in order to provide explanation, one gets an explanation-based system. Note that determining good explanation can be tricky especially for global constraint with tightly tailored algorithms. One need to get back to the proof of the algorithms in order to provide interesting explanations. However, our experience shows that sticking with very simple explanations can still give very interesting results [15].

That last statement has recently led us to design a new explanation mechanism that would not need to modify the inner definitions of provided constraints but rather provide a generic way of computing explanations following patterns.

3 Using explanations for constraint removal

Now that we have an explanation-based constraint system, it is time to see what can be done with it. Removing a constraint⁵ needs two main steps:

- *getting back values*: values whose removal were directly or not due to the removed constraint need to be put back in their respective domain;
- *reaching consistency*: newly restored values could possibly be done in another way. A consistency-check is to be done in order to get back to locally coherent state as if the removed constraints never appeared in the constraint system.

3.1 Getting values back

The first step of a constraint removal is quite easy since all the concerned values are those whose eliminating explanation contains the removed constraints. In order to achieve that behavior efficiently, we maintain in each constraint c the set of eliminating explanations which contains c . When getting back values, those sets need to be updated for each undone value removal.

3.2 Repropagating

The second step of a constraint removal relies upon checking all related constraints when restoring values in the domain of a variable. The easy way is to completely check the constraints but simple functions handling value restoration can be easily written: for example, when the upper bound of the second variable of a constraint $X \geq Y + c$ is increased (value restoration), to check if that new value is compatible with the constraint only needs to pretend that the upper bound of the first variable has been modified which gives the code of Fig. 8 in the PaLM system.

```
[awakeOnRestoreSup(c: PalmGreaterOrEqualxyc, idx: integer) : void
-> if (idx = 2) awakeOnSup(c, 1)]
```

Fig. 8. Example of repropagation under PaLM

⁵ Constraint removal in dynamic problems has already been studied [2, 5] but here we simplify the algorithms thanks to explanations [13].

4 Explanation-based constraint programming

Explanation-based constraint systems can be used in various situations. The following two sections detail some of them.

4.1 High-level usage of explanations

- *Handling dynamic problems*

As we saw, explanation are useful for dynamically adding or removing constraints before/during or after search.

- *Providing information to the user*

Explanations are a wonderful tool for giving the user information such as: how come I do not get value x for variable v , how come that problem has no solution, how come y is only remaining possible value for variable v ? ...

- *Handling over-constrained problems*

The combination of the two previous usages of explanation-based system lead to automatic (or user-guided) constraint relaxation system [12]. The main interest of using such an approach is it does not need to know all the constraints weights before the beginning of the search.

4.2 Low-level usage of explanations

More low-level usages of explanation-based search come to mind: as search is often based on early discovery of dead-ends in order to focus on feasible parts of the search tree, why not use explanations as a hint for future explorations or for pointing out erroneous previous choices.

Actually, the **PaLM** system allows an easy constraints addition or removal while solving a problem. That feature allows us to get rid of backtracking-based complete search algorithms. For example, ideas such as *dynamic-backtracking* [7] can be embedded in our system to provide that new way of solving constraint problems.

A new search paradigm The first step to get in that new search paradigm is to consider making a choice during the search⁶ as a mere constraint addition. As constraints are added, a contradiction may occur.

In such a situation, there is no need for backtracking: simply consider the nogood that explains the contradiction (calling `self_explain(fv, DOM, e)` for example if `fv` is the failing variable) and select a constraint in it. As shown in [13] which was derived from [7], in order to remain complete, one needs to select the more recent choice constraint in the nogood. In order to move from that dead end, one can remove the considered constraint and add its negation (simulating setting aside a testing value for CSP).

```

[ solve(pb: PalmProblem): boolean
-> unassignedVars := pb.vars,
  try (
    while (size(unassignedVars) != 0) (
      let idx := nextVarToAssign(pb),
          v := unassignedVars[idx],
          a := selectValToAssign(pb, v)
      in (
        try (
          unassignedVars :delete v,
          post(pb, v == a, 0),
          propagate(pb)
        )
        catch (PalmContradiction) ( // An empty domain has been found
          handleContradiction()
        )
      )
    ),
    true // A solution was found
  )
  catch (contradiction) ( // A choco contradiction means that
    false // there is no solution
  )
]

```

Fig. 9. The PaLM code for exploring the search space

```

[ handleContradiction(): void
-> if knownFailingVariable?() ( // A failure occurred ?
  let e := PalmSet()
  in (
    self_explain(getFailingVariable(), DOM, e), // Computing a nogood
    if (empty(e)) // The problem itself is unsolvable
      contradiction!() // raise a choco contradiction
    else (
      let ct := selectConstraint(e)
      in (
        if known?(ct) (
          unassignedVars :add ct.v1,
          remove(ct), // Relaxing the constraint and removing its effects

          try (
            e :delete ct,
            // posting the associated negation constraint
            post(CURRENT_PB, ct.v1 != ct.cste, e),
            propagate(CURRENT_PB) // restoring consistency
          )
          catch PalmContradiction (
            handleContradiction() // Doing it recursively
          )
        )
        else ( // no choice constraint appear in the nogood
          contradiction!()
        )
      )
    )
  ))) ]

```

Fig. 10. The PaLM code for handling a contradiction

A search for explanation-based systems Search for explanation-based systems can be made in a classical way (see figure 9): assignments constraints are added step by step. When a failure occurs, an explanation is computed and an assignment constraint is chosen for removal (it was a misleading choice). In order to avoid unwanted loops in the search, the negation of the removed constraint is added (the tested value is rejected). Figure 10 shows such a contradiction handling mechanism.

Of course, that new constraint can remain in the system only if all the other constraints appearing in the nogood remain active. The **PaLM** system provides tools to deal with that kind of constraint (called **indirect constraints**): an indirect constraint keeps its validity context (the set of the other constraints in the nogood that provoked the adding) and the **PaLM** system makes sure that as soon as one of those constraints is removed, all its depending constraints are also removed. When that new constraint is added, a failure may happen, hence it must be treated recursively.

As we can see, that contradiction handling mechanism appears as a full replacement of the standard backtracking schema in *classical* search. The information that can be brought from failures is used in that mechanism.

A point still needs attention. Indeed, when a failure occurs, the propagation queue may not be empty: the pending events may stay valid even if a previous choice is dismissed. If as usual when encountering a contradiction, that propagation queue is emptied, those events will not be propagated possibly losing consistency. In our approach, the queue must not be cleaned in order to remain consistent.

Variations around a same theme We developed two sets of searches: complete algorithms (that keep most information) and incomplete methods (that tend to forget information early). Our first attempt was to develop an intelligent backtrack based on solver given explanation for solving scheduling problems [10].

First results obtained with that intelligent backtracker (we were able to optimally solve a problem from the literature for the first time) led us to go further and fully integrate constraint propagation in a dynamic-backtracking like algorithm leading to two main algorithms: **mac-dbt** [13] a complete search method for solving CSP and **path-repair** [15] an incomplete search method.

The first one (**mac-dbt**) uses the previous search algorithm and is a search algorithm for discrete CSP. Our experiments comparing it to **mac** showed that it could greatly outperform it when dealing with structured problems. A preliminary version of that algorithm was developed for numeric CSP [14] showing the same results: solving structured problem is beneficial for our method compared to other algorithms such those of Numerica [11].

The second algorithm that we developed using an explanation-based solver is the **path-repair** algorithm [15]. It is a local search technique working on partial

⁶ In classical CSP, this means giving a value to a variable. For numerical CSP, this means splitting a domain. For scheduling problems, this can be adding a precedence constraint between tasks ...

assignments that uses filtering technique to prune the search space. It can be seen as a modified version of `mac-dbt`: the choice of the value assignment to undo is left to a heuristic (we loose completeness but are more able to focus on *annoying* constraints). First results on scheduling problems (namely open-shop problems) show that our general algorithm competes well with highly specialized recent tabu searches. Moreover, we recently solved for the first time three open problems from the literature (problems presented in [9]).

5 The PaLM suite

The **PaLM** system is in fact made of several modules giving a real suite dedicated to explanation-based constraint programming. The main module is the **PaLM** core file which handles variables described by their upper and lower bounds. The **PaLMenum** module handles classical variables for CSP and the **PaLMschedule** module is dedicated to scheduling problems.

5.1 PaLM

The core system of the suite contains all the necessary tools for handling explanations (those tools are shared by all the modules). Variables in the **PaLM** core file are described by their upper and lower bounds (during propagation no holes can be made in the domains).

Classical unary and binary arithmetical constraints are provided as well as a global constraint handling large linear combinations of variables.

The **PaLM** system provides several tools through its API:

- object declaration methods
`makePalmProblem`, `makePalmIntVar`, ...
- constraint posting methods
classical posting `post(pb, ct)`, weighted posting `post(pb, ct, w)` and indirect posting `post(pb, ct, e)`
- operators redefinition for easily defining constraints
`==`, `!=`, `<=`, ...
- global constraints for high level constraint programming
`alldifferent`, general linear combinations of variables, ...

5.2 PaLMenum

PaLMenum is a sub-module dedicated to variables whose domain is described as a list of values (holes can be made in it).

The same basic constraints as in the **PaLM** core file have been provided in that sub-module. Moreover, constraints described as tuples of authorized values are to be integrated in that module. Such a constraint corresponds to a propagation schema. As for now **PaLMenum** provided **ac4**-like constraints, **ac6**-like constraints and **pic**-like⁷ constraints.

⁷ PIC stands for Path-Inverse Consistency [6].

The **PaLM** API has been extended to be used with **PaLMenu** (for example a `makePalmEnumIntVar` method is provided). Enumerated domains specific constraints are given: a specialized version of `alldifferent`, the `element` constraint, ...

5.3 PaLMschedule

PaLMschedule is a **PaLM** extension dedicated to scheduling. The variables to be manipulated are tasks in this module. As for now, unary resources for non-preemptive problems are handled using task-intervals [4] and the associated edge finders.

The **PaLMschedule** API has the following features

- object declaration
`makePalmScheduleProblem`, `makePalmUnaryResource`, `makeTask`,
`makePalmTaskInterval`
- constraint definition and posting
operator `-->` for defining precedences, `post(pb, resource)` for posting in one instruction necessary constraints for handling unary resources.

A specific application for solving open-shop problems has been designed. It fully implements ideas presented in [10, 8]

6 Conclusion

In this paper, we presented the **PaLM** suite: a set of explanation-based constraint solvers. We showed how to implement such a system on top of a *classical* constraint programming system. Moreover, we presented what could be done with the **PaLM** suite and detailed some experimental results obtained using that system.

Our current works include: extending **PaLMenu** (with high order consistency techniques) and **PaLMschedule** (with cumulative constraints), providing new explanations mechanisms, using *k*-relevance [1], developing applications using the **PaLM** suite ...

Acknowledgments

The authors would like to thank François Laburthe for providing us that great tool that `choco` is and Romuald Debruyne for his insights for implementing enumerated-domains variables and constraints.

References

- [1] Roberto J. Bayardo Jr. and Daniel P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *AAAI'96*, 1996.

- [2] Christian Bessière. Arc consistency in dynamic constraint satisfaction problems. In *Proceedings AAAI'91*, 1991.
- [3] Yves Caseau, François-Xavier Josset, and François Laburthe. Claire: combining sets, search and rules to better express algorithms. In D. De Schreye, editor, *Proc. of the 15th International Conference on Logic Programming, ICLP'99*, pages 245–259. MIT Press, 1999.
- [4] Yves Caseau and François Laburthe. Improving clp scheduling with task intervals. In P. Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming, ICLP'94*, pages 369–383. MIT Press, 1994.
- [5] Romuald Debruyne. Arc-consistency in dynamic CSPs is no more prohibitive. In *8th Conference on Tools with Artificial Intelligence (TAI'96)*, pages 299–306, Toulouse, France, 1996.
- [6] E. Freuder and C. Elfe. Neighborhood inverse consistency preprocessing. In *Proceedings of AAAI'96*, pages 202–208, Portlan, OR, 1996.
- [7] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
- [8] Christelle Guéret and Narendra Jussien. Combining AI/OR techniques for solving open shop problems. In *Workshop on Integration of Operations Research and Artificial Intelligence techniques in Constraint Programming (CP-AI-OR)*, Ferrara, Italy, 25–26 February 1999.
- [9] Christelle Guéret and Christian Prins. A new lower bound for the open-shop problem. *AOR (Annals of Operations research)*, 92:165–183, 1999.
- [10] Christelle Guéret, Narendra Jussien, and Christian Prins. Using intelligent backtracking to improve branch and bound methods: an application to open-shop problems. *European Journal of Operational Research*, page to appear, 2000.
- [11] P. Van Hentenryck, P. Michel, and L. Deville. *Numerica, a modeling language for global optimization*. MIT press, 1997.
- [12] Narendra Jussien and Patrice Boizumault. Best-first search for property maintenance in reactive constraints systems. In *International Logic Programming Symposium*, pages 339–353, Port Jefferson, N.Y., USA, October 1997. MIT Press.
- [13] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Sixth international conference on principles and practice of constraint programming (CP'2000)*, September 2000.
- [14] Narendra Jussien and Olivier Lhomme. Dynamic domain splitting for numeric csp. In *European Conference on Artificial Intelligence*, pages 224–228, Brighton, United Kingdom, August 1998.
- [15] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. In *Seventh National Conference on Artificial Intelligence – AAAI'2000*, pages 169–174, Austin, TX, USA, August 2000.
- [16] François Laburthe. Choco: implementing a cp kernel. In *CP00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems (TRICS)*, Singapour, September 2000.