# Investigating Parallel Interpretation-Tree Model Matching Algorithms with ProSet-Linda*

W. Hasselbring
Dept. of Computer Science, University of Dortmund
Informatik 10 (Software Technology), D-44221 Dortmund, Germany
Telephone: 49-(231)-755-4712, Fax: 49-(231)-755-2061
email: willi@ls10.informatik.uni-dortmund.de

R. B. Fisher
Dept. of Artificial Intelligence, University of Edinburgh
5 Forrest Hill, Edinburgh EH1 2QL, Scotland, United Kingdom
Telephone: 44-(31)-650-3098, Fax: 44-(31)-650-6899
email: rbf@aifh.ed.ac.uk

**Abstract**

This paper discusses the development of algorithms for parallel interpretation-tree model matching for 3-D computer vision applications such as object recognition. The algorithms are developed with a prototyping approach using ProSet-Linda. ProSet is a procedural prototyping language based on the theory of finite sets. The coordination language Linda provides a distributed shared memory model, called tuple space, together with some atomic operations on this shared data space. The combination of both languages, viz. ProSet-Linda, is designed for prototyping parallel algorithms.

The classical control algorithm for symbolic data/model matching in computer vision is the *Interpretation Tree* search algorithm. This algorithm has a high computational complexity when applied to matching problems with large numbers of features. This paper examines parallel variations of this algorithm. Parallel execution can increase the execution performance of model matching, but also make feasible entirely new ways of solving matching problems. In the present paper, we emphasize the *development* of parallel algorithms with a prototyping approach, not the presentation of performance figures displaying increased performance through parallel execution. The expected improvements attained by the parallel algorithmic variations for interpretation-tree search are analyzed.

The implementation of ProSet-Linda is briefly discussed.

**Keywords:** model-based vision, object recognition, parallel search, prototyping parallel algorithms.

*This is Software-Technik Memo Nr. 77, University of Dortmund, and DAI Research Paper No. 722, University of Edinburgh, December 1994.

# Contents

# 1 Introduction

Three-dimensional computer vision is commonly divided into several levels. In the research investigated at Edinburgh, low-level vision is concerned with processing range data acquired by a laser range scanner to eliminate noise [9]. Medium-level vision is concerned with identifying geometric surfaces [29]. High-level vision tries, for example, to identify the shape and position of data objects using matched given model features. In the high-level components, first the model invocation process pairs likely model and data features for further consideration [5]. Model matching then uses the candidate matches proposed by the invocation to form consistent groups of matches.

The classical control algorithm for symbolic model matching in computer vision is the *Interpretation Tree* search algorithm, as used by Grimson and Lozano-Perez [12]. The algorithm searches a tree of potential model-to-data correspondences, such that each node in the tree represents one correspondence and the path of nodes from the current node back to the root of the tree is a set of simultaneous pairings. This model matching algorithm is a specialized form of the general AI tree search technique, where branches are pruned according to a set of consistency constraints according to some (geometric) criterion. The goal of the search algorithm is to maximize the set of consistent model-to-data correspondences in an efficient manner. Finding these correspondences is a key problem in model-based vision, and is usually a preliminary step to object recognition, pose estimation, or visual inspection.

Unfortunately, this algorithm has the potential for combinatorial explosion. To reduce the complexity, techniques for pruning the trees have been developed, thus limiting the number of candidate matches considered. The main technique commonly used is based on *pruning constraints* [12] (which locally reject pairings that are inconsistent, and hence eliminate all of the search that might further extend this inconsistent pairing) and *early termination* [11]. The latter stops search when a given number of pairings (a threshold) is reached. However, even with these effective forms of pruning, the algorithms still can have exponential complexity, making it unsuitable for use in scenes with many features.

Parallel execution can increase the execution performance of model matching, but also investigate entirely new ways of solving matching problems. As has been observed [2], it is only from new algorithms that orders of magnitude improvements in the complexity of a problem can be achieved:

> "An idea that changes an algorithm from $n^2$ to $n \log n$ operations, where $n$ is proportionate to the number of input elements, is considerably more spectacular than an improvement in machine organization, where only a constant factor of run-time is achieved." [2, page 250]

Thus, rapid prototyping of parallel algorithms may serve as the basis for developing parallel, high-performance applications. **In this paper, we present a methodology for the *development* of parallel high-level vision algorithms using a ProSet-Linda based prototyping approach.**

Parallelism in low- and medium-level computer vision is usually programmed in a *data-parallel* way, for instance based on the computational model of cellular automata [14]. For high-level symbolic computer vision, the data-parallel approach is not appropriate, as symbolic computations have an irregular control flow depending on the actual input data. The underlying model of the data-parallel approach uses synchronous communication. The programmer often has to think in *simultaneities* while constructing a program, because she or he often has to focus on more than one process at a time. This complicates parallel programming significantly.

Data parallelism is opposed to *control parallelism*, which is achieved through multiple threads of control, operating independently. The data-parallel approach lets programmers replace iteration (repeated execution of the same set of instructions with different data) with parallel execution. It does not address a more general case, however: performing many interrelated but *different* operations at the same time. This ability is essential in developing algorithms for high-level symbolic computer vision.

Developing parallel algorithms is in general considered an awkward undertaking. The goal of the ProSet-Linda approach is to partially overcome this problem by providing a tool for prototyping parallel algorithms [15]. To support prototyping parallel algorithms, a prototyping language should provide simple and powerful facilities for dynamic creation and coordination of parallel processes. Process communication and synchronization in ProSet-Linda is reduced to concurrent access to a shared data pool, thus relieving the programmer from the burden of having to consider all process inter-relations explicitly. The parallel processes are decoupled in time and space in a simple way: processes do not have to execute at the same time and do not need to know each other's addresses (as it is necessary with message-passing systems). The shared data pool in the Linda concept is called *tuple space*, because its access unit is the tuple, similar to tuples in ProSet (see Section 2.1).

Section 2 gives a brief introduction to the tool for implementing the parallel variations of the interpretation-tree tree search algorithm, viz. ProSet-Linda for prototyping parallel algorithms. Section 3 describes the standard interpretation-tree algorithm. Section 4 takes a general look at parallel interpretation-tree search. We do not parallelize the standard interpretation-tree algorithm, but the non-wildcard and best-first alternatives in Sections 5 and 6, respectively. Section 7 briefly discusses the implementation of ProSet-Linda. Section 8 draws some conclusions. The programs are programmed literately. Appendix A presents a brief introduction to literate programming with `noweb`.

## 2 Prototyping Parallel Algorithms with ProSet-Linda

Before presenting the implementation of the parallel interpretation-tree model matching algorithms, we have a look at ProSet-Linda as the language used for implementation. The procedural, set-oriented language ProSet [4] is a successor to SETL [26]. ProSet is an acronym for PROtotyping with SETs. The high-level structures that ProSet provides qualify the language for prototyping. Refer to [3] for a full account of prototyping with set-oriented languages. A case study for prototyping using SETL is documented in [20]. The use of SETL for prototyping algorithms for parallelizing compilers is described in [23].

Section 2.1 introduces the basic concepts of ProSet and Section 2.2 gives a short description of the features for parallel programming.

### 2.1 Basic Concepts

ProSet provides the data types atom, integer, real, string, Boolean, tuple, set, function, module, and instance. Modules may be instantiated to obtain module instances. It is a *higher-order* language, because functions and modules have first-class rights. ProSet is weakly typed, i.e., the type of an object is in general not known at compile time. Atoms are unique with respect to one machine and across machines. They can only be created and compared for equality. Tuples and sets are compound data structures, the components of which may have different types. Sets are unordered collections while tuples are ordered. There is also the undefined value `om` which indicates undefined situations.

As an example consider the expression `[123, "abc", true, {1.4, 1.5}]` which creates a tuple consisting of an integer, a string, a Boolean, and a set of two reals. This is an example of what is called a *tuple former*. As another example consider the *set forming* expression `{2*x: x in [1..10] | x>5}` which yields the set `{12, 14, 16, 18, 20}`. Sets consisting only of tuples of length two are called maps. There is no genuine data type for *maps*, because set theory suggests handling them this way.

The control structures have ALGOL as one of its ancestors. There are `if`, `case`, `loop`, `while`, and `until` statements as usual, and the `for` and `whilefound` loops which are custom tailored for iteration over compound data structures. The quantifiers (∃, ∀) of predicate calculus are provided.

### 2.2 Parallel Programming

In ProSet, the concept of process creation via Multilisp's *futures* [13] is adapted to set-oriented programming and combined with the coordination language Linda [10] to obtain the parallel programming language ProSet-Linda. Linda is a coordination language which provides means for synchronization and communication through so-called *tuple spaces*. These tuple spaces are virtual shared data spaces accessed by an associative addressing scheme. Synchronization and communication in ProSet-Linda are carried out through several atomic operations: addition, removal, reading, and updates of individual tuples in tuple space. Linda and ProSet both provide tuples; thus, it is quite natural to combine both models to form a tool for prototyping parallel algorithms.

The access unit in tuple space is the tuple. A tuple space may contain any number of copies of the same tuple: it is a multiset, not a set. Process communication and synchronization in Linda is called *generative communication*, because tuples are added to, removed from, and read from tuple space concurrently. Synchronization is done implicitly. Reading access to tuples in tuple space is associative and not based on physical addresses, but rather on their expected content described in *templates*. This method is similar to the selection of entries from a data base. Refer to [1] for a full account of programming with Linda. ProSet supports multiple tuple spaces. Atoms are used to identify tuple spaces uniquely.

Multisets are a powerful data structure for parallel programming. Since a tuple space is a multiset of tuples, it may contain multiple copies of a tuple, whereas in a set each element exists exactly once. Because of concurrent access by cooperating processes to a tuple space, it is necessary to have multisets and not sets for coordination. Multisets are, therefore, a good basis for communication between cooperating processes, because the data flow is not restricted unnecessarily. Furthermore, multisets are dynamic data structures that alleviate the treatment of dynamically varying size problems. The benefit of using multisets is the possibility of describing compound data without any form of constraint or hierarchy between its components. This is also the case for sets, but not for data structures such as ordered lists which impose an ordering on the elements. Consequently, multisets allow a high degree of parallelism for cooperating processes.

Several ProSet-Linda library functions are provided for handling multiple tuple spaces dynamically. The function `CreateTS(limit)` creates a new tuple space and returns its identity (an atom). Since one can have exclusive access to a fresh tuple-space identity, `CreateTS` supports information hiding. As mentioned above, atoms are unique for one machine and across machines. The integer parameter `limit` specifies a limit on the expected or desired size of the new tuple space. It provides an indication of the total number of passive and active tuples which are allowed in a tuple space concurrently. `CreateTS(om)` would instead indicate that the expected or desired size is not limited. The function `ExistsTS(TS)` yields `true`, if `TS` is an atom that identifies an existing tuple space, it is `false` otherwise. The function `ClearTS(TS)` removes all tuples from the specified tuple space. The function `RemoveTS(TS)` calls `ClearTS(TS)` and removes `TS` from the list of existing tuple spaces.

ProSet provides three tuple-space operations. The `deposit` operation deposits a tuple into a tuple space:

```
deposit [ "pi", 3.14 ] at TS end deposit;
```

`TS` is the tuple space at which the tuple `[ "pi", 3.14 ]` has to be deposited. The `fetch` operation tries to fetch and remove a tuple from a tuple space:

```
fetch ( "pi", ? x ) at TS end fetch;
```

The template `( "pi", ? x )` only matches tuples with the string `"pi"` in the first and anything in the second field. The templates are enclosed in parentheses and not in brackets in order to set the templates apart from tuples. The optional *l*-values specified in the formals (the variable `x` in our example) are assigned the values of the corresponding tuple fields, provided matching succeeds. Formals are prefixed by question marks. The selected tuple is removed from tuple space. Another example for a `fetch` operation with a single template follows:

```
fetch ( "name", ? x |(type $(2) = integer) ) at TS end fetch;
```

This template only matches tuples with the string `"name"` in the first field and integer values in the second field. The symbol `$` may be used like an expression as a placeholder for the values of corresponding tuples in tuple space. The expression `$(i)` then selects the `i`th element from these tuples. Indexing starts with `1`. It is only allowed to use the symbol `$` this way in expressions that are part of templates. As usual in ProSet, `|` means *such that*. The Boolean expression after `|` may be used to *customize* matching by restricting the set of possibly matching tuples. The selected tuple is removed from tuple space. If no `else` statements are specified as in the above example then the statement suspends until a match occurs. If statements are specified for the selected template, these statements are executed. An example with multiple templates, associated statements, and an `else` statement follows:

```
fetch ( "name", ? x |(type $(2) = integer) ) => put("Integer fetched");
  xor ( "name", ? x |(type $(2) = set) )     => put("Set fetched");
   at TS
 else put("Nothing fetched");
end fetch;
```

This statement fetches at most one tuple. The `meet` operation is the same as `fetch`, but the tuple is not removed and may be changed:

```
meet ( "pi", ? x ) at TS end meet;
```

Changing tuples is done by specifying expressions `into` which specific tuple fields will be changed. Consider

```
meet ( "pi", ? into (2.0 * $(2)) ) at TS end meet;
```

where the value of the second element of the met tuple is doubled. This statement changes at most one tuple. Tuples which are met in tuple space may be regarded as shared data since they remain in tuple space irrespective of changing them or not. For a detailed discussion of prototyping parallel algorithms in set-oriented languages refer to [17].

# 3 The Standard Interpretation-Tree Algorithm

Consider a set $\{ d_i \}$ of $D$ data features and a set $\{ m_i \}$ of $M$ model features. The root of the interpretation tree has no pairings. The first level expands the root node to pair all of the $M$ model features with data feature $d_1$. The second level in the tree expands each of these nodes to pair all model features with data feature $d_2$ (multiple use of a given $m_i$ is allowed), and so on. The expansion continues for all $D$ data features. At each node at level $k$ in the tree, therefore, there is a hypothesis with $k$ features matched. Figure 1 displays an example.

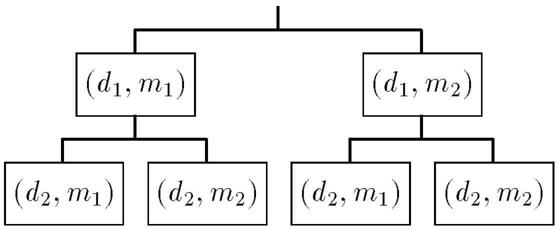


Figure 1: An example for the standard interpretation tree with data features $\{d_1, d_2\}$ and model features $\{m_1, m_2\}$.

If this interpretation tree is explored completely, there are $M^D$ "leaf" nodes (complete interpretations) at the bottom of the tree and

$$\sum_{i=0}^{D} M^i = \frac{M^{D+1} - 1}{M - 1} \doteq M^D$$

nodes in the full tree. If either $M$ or $D$ are of any reasonable size (e.g. larger than 5 as is usual in practical cases), then we can expect to have excessively large search trees.

An additional complication is that one usually wishes to include at each level of the tree a "wildcard" feature that will match with any other feature. This is necessary because it may not always be possible to find a model feature that matches a given data feature at the current level of the tree (because of fragmentation, bad segmentation, noise, unrelated features, occlusion, etc.). This increases the number of leaf nodes to $(M + 1)^D$.

One way to reduce the amount of searching is to 'prune whole branches of the tree', by showing that a given pairing or sequence of pairings is inconsistent. In consequence, all descendents from that node in the tree will also be inconsistent and need not be explored. The most common approach uses unary and binary pruning constraints. *Unary* constraints eliminate model-to-data pairings when some shared property is inconsistent (see also Appendix B). *Binary* constraints eliminate hypotheses when a relative property between a pair of model features is inconsistent with the same property between the corresponding pair of data features (see also Appendix C).

When wildcards are allowed, examination of the search process shows there are several sources of wasted effort. The algorithm could then accept an exponential number of correctly matchable features. One key term is $2^C$, arising from the power set of the $C$ matchable features. This complexity occurs because each matchable data feature can be either matched with the correct model feature or the wildcard. Examination of a typical search tree shows that most of the tree consists of paths containing either members of this power set or many wildcards. Another source of wasted effort is the re-exploration of identical subtrees under each initial set of matches. A number of matching algorithms which reduce this wasted exploration have been developed [7].

# 4 Parallel Interpretation-Tree Search

Parallelism in a tree search algorithm can be obtained by searching the branches of the tree in parallel. A simple approach would be to spawn a new process for each subtree to be evaluated. This approach would not work well since the amount of parallelism is determined by the input data and not by, for instance, the number of available processors.

The programs which will be presented in the following sections are master-worker applications (also called *task farming*). In a master-worker application, the task to be solved is partitioned into independent subtasks. These subtasks are placed into a tuple space, and each process in a pool of identical workers then repeatedly retrieves a subtask description from the tuple space, solves it, and puts the solutions into the tuple space. The master process then collects the results. An advantage of this programming approach is easy load balancing because the number of workers is variable and may be set to the number of available processors.

Similar to seqential tree search, it is in general not necessary to search the entire tree: *bounding rules* avoid searching the entire tree. For interpretation-tree model matching, the bounding rules are defined by geometric constraints (see Appendix C) and termination thresholds to prune entire subtrees.

# 5 Parallel Non-wildcard Search Tree Algorithms

We will now discuss parallel variations of the sequential non-wildcard search tree algorithm, which is introduced in Section 5.1. Section 5.2 presents a parallel non-wildcard complete search tree algorithm which finds all satisfactory matches. A match is satisfactory when the termination number of matched features has been reached.

The sequential non-wildcard search tree algorithm stops when the first satisfactory match has been found. It does not search for *all* solutions. Section 5.3 presents a parallel non-wildcard search tree algorithm which stops, when the first satisfactory match has been found. This algorithm is quite similar to the sequential non-wildcard search tree algorithm, but the tree is searched in a non-deterministic order and *not* depth-first following the leftmost branches first.

### 5.1 The Sequential Non-wildcard Search Tree Algorithm

As many of the nodes in the standard interpretation tree algorithm arise because of the use of wildcards, an alternative search algorithm explores the same search space, but it does not use a wildcard model feature to match otherwise unmatchable data features [6]. The tree in Figure 2 displays an example non-wildcard interpretation tree. With the sequential algorithm, the tree is searched depth-first following the leftmost branches first (no pruning is shown here to illustrate the shape of the tree).

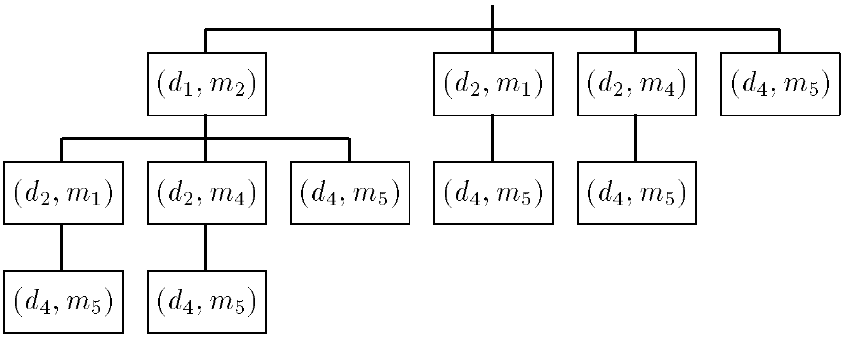


Figure 2: An example for the non-wildcard interpretation tree for $\Omega = [s_1, s_2, s_3, s_4] = [(d_1, m_2), (d_2, m_1), (d_2, m_4), (d_4, m_5)]$.

The essence of the difference between the standard interpretation-tree and the non-wildcard interpretation-tree algorithm is that the search process skips over all data pairings that use a wildcard, to consider the next true data-model feature pairing. This results in a flattening of the search tree. The algorithm has two phases:

1. The tuple $\Omega = [s_k] = [(d_{i(k)}, m_{j(k)})], k = 1..N$ of all pairs of features satisfying the unary pairing constraints is formed, such that if $s_r$ is before $s_s$ (i.e. $r < s$), then $i(r) \leq i(s)$, and if $i(r) = i(s)$, then $j(r) < j(s)$.

   With the non-wildcard interpretation-tree algorithm, the ordering is determined by the index numbers of the data features. With the best-first interpretation-tree algorithm, the ordering is determined by the plausibilities (see Section 6.1 and Appendix B).

2. The search tree is explored such that each extension of a branch is formed by appending new entries from $\Omega$, subject to the constraints that (1) each data feature appears at most once on a path through the tree and (2) the data features are used in order (with gaps allowed).

## 5.2 Parallel Complete Search Tree Algorithm

This section presents a parallel implementation of the non-wildcard complete search tree algorithm given in Section 5.1 which provides all satisfactory matches.

The main program for the parallel non-wildcard complete search tree model matching is the master process:

⟨*Non-wildcard complete search tree main program* 8a⟩≡
```
  program Complete;
    ⟨Tuple space declarations 8b⟩
    ⟨Hypotheses from the model invocation 8c⟩
  begin -- The master (main program):
    ⟨Get the number of worker processes 8d⟩
    ⟨Get the termination threshold 9a⟩
    ⟨Spawn the worker processes 9b⟩
    ⟨Deposit the initial task tuples 9c⟩
    ⟨Initialize the number of finished workers 9d⟩
    ⟨Let the worker processes start working 9e⟩
    ⟨Wait for the workers to finish 9f⟩
    ⟨Fetch the results for complete search 10a⟩

    -- The procedure declarations:
    ⟨Worker procedure for non-wildcard search 10b⟩
    ⟨Procedure for reading the hypotheses from the invocation 32⟩
  end Complete;
```

In contrast to Pascal or Modula, in ProSet the main program precedes the procedure declarations to support a top-down presentation.

This paper has been processed with the literate programming tool `noweb` to implement and present the program code. Refer to Appendix A for a brief introduction to reading literate programmed code which has been written with `noweb`. In summary, this approach allows one to associate code and descriptive text in the same document, and then extract the code portion to create executable text.

We use two tuple spaces. One for the work tasks and one for the results:

⟨*Tuple space declarations* 8b⟩≡
```
    visible constant WORK := CreateTS(om),     -- for the work tasks
                     RESULT := CreateTS(om);  -- for the results
```

The hypotheses from the model invocation are read by the procedure `GetHypos` from standard input (see Appendix B). These form the initial set $\Omega$.

⟨*Hypotheses from the model invocation* 8c⟩≡
```
    visible constant hypos := GetHypos (); -- visible for the worker processes
```

Note that the plausibilities from the invocation are not considered with non-wildcard tree search. For simplicity the procedure `GetHypos` is used for both the non-wildcard tree search and the best-first tree search. The latter uses the plausibilities (see Section 6).

The number of worker processes is an argument to the main program. This could be, for instance, the number of available processors:

⟨*Get the number of worker processes* 8d⟩≡
```
    NumWorker := argv(2); -- This is a string. Convert it to an integer via C's atoi:
    NumWorker := c_fct_call atoi (NumWorker : c_string) c_integer;
```

The termination threshold for satisfactory matches is the next argument to the main program. It has to be less than or equal to the number of data features:

⟨*Get the termination threshold* 9a⟩≡
```
Threshold := argv(3); -- This is a string. Convert it to an integer via C's atoi:
Threshold := c_fct_call atoi (Threshold : c_string) c_integer;
if (Threshold < 2) ∨ (#{x(1): x ∈ hypos} < Threshold) then
  put("The termination threshold is not acceptable!");
  stop;
end if;
```

The `#` operator returns the number of elements contained in a compound data structure. Note, that in a set each contained element exists exactly once.

The master spawns `NumWorker` worker processes to do the work:

⟨*Spawn the worker processes* 9b⟩≡
```
for i ∈ [1..NumWorker] do
  || closure Worker (i, Threshold); -- Spawn the worker processes
end for;
```

The || operator spawns a new process. The `closure` operator assures that the spawned procedure `Worker` has no side effects on global variables. See [17] for details.

The master puts the initial task tuples into tuple space `WORK`:

⟨*Deposit the initial task tuples* 9c⟩≡
```
for Entry ∈ hypos do
  deposit [ {Entry} ] at WORK end deposit;
end for;
```

For the example tree of Figure 2, these initial task tuples are (the plausibilities from the invocation are not displayed here):

```
[{[1,"m_2"]}], [{[2,"m_1"]}], [{[2,"m_4"]}], [{[4,"m_5"]}]
```

These initial tasks are the nodes at the first level of the interpretation-tree.

After depositing the initial task tuples, the master initializes a shared counter for the number of finished workers at tuple space `RESULT`:

⟨*Initialize the number of finished workers* 9d⟩≡
```
deposit [ "Finished Workers", 0 ] at RESULT end deposit;
```

After initializing the tuple spaces, the workers are enabled to start their work:

⟨*Let the worker processes start working* 9e⟩≡
```
deposit [ "start", "now" ] at WORK end deposit; -- Start the workers
```

Alternatively, we could omit the `[ "start", "now" ]` tuple entirely and spawn the workers directly after depositing the initial tasks (and let them start working immediately). However, for testing with small input data sets, it is useful to defer the workers.

After spawning the workers and initializing the tasks, the master waits until all workers have done their work (by executing a blocking fetch until the number of finished worker processes equals `NumWorker`):

⟨*Wait for the workers to finish* 9f⟩≡
```
fetch ( "Finished Workers", NumWorker ) at RESULT end fetch;
```

Then the master fetches the possible matches from tuple space RESULT and writes the results to standard output:

⟨*Fetch the results for complete search* 10a⟩≡

```
loop
  fetch ( ? Match ) at RESULT
   else quit; -- No more results: quit the loop
  end fetch;
  put("Match = ", Match);
end loop;
```

For the example tree of Figure 2 with Threshold equal to 3 the program prints out (the plausibilities from the invocation are not displayed here):

```
Match = { [1,"m_2"],[2,"m_4"],[4,"m_5"] }
Match = { [1,"m_2"],[2,"m_1"],[4,"m_5"] }
```

This was the implementation of the master process (the main program). Now let us look at the worker procedure. Each worker first waits to be enabled and then executes in an endless loop:

⟨*Worker procedure for non-wildcard search* 10b⟩≡

```
procedure Worker (i, Threshold);
begin
  ⟨Wait to be enabled 10c⟩
  loop
    ⟨Fetch task 10d⟩
    ⟨Evaluate task for non-wildcard search 11b⟩
  end loop;

  ⟨Procedure for consistency check 36a⟩
  ⟨Auxiliary procedure inc2 11a⟩
end Worker;
```

To become enabled each worker waits to *meet* the tuple ["start", "now"]:

⟨*Wait to be enabled* 10c⟩≡

```
    meet ( "start", "now" ) at WORK end meet;
```

Each worker first checks whether there are more task tuples in tuple space WORK, and terminates when there is no more work to do:

⟨*Fetch task* 10d⟩≡

```
     fetch (? MyPath) at WORK
      else -- increase the number of finished workers:
        meet ( "Finished Workers", ? into closure inc2($) ) at RESULT end meet;
        return;
     end fetch;
```

Before termination, the shared counter "Finished Workers" in tuple space RESULT is incremented to indicate the termination to the master.

In principle it should be possible to write simply "into $(2)+1" within the above meet operation. For a tuple T, the expression T(i) selects the ith element from T. Indexing starts with 1. Unfortunately, the current version of the ProSet compiler only accepts simple functions calls of the form "`closure inc2($)`" after `into` and `|` in templates. Therefore, we have to write the auxiliary function `inc2`:

⟨*Auxiliary procedure inc2* 11a⟩≡
```
procedure inc2 (x); begin
    return x(2)+1;
end inc2;
```

This syntactical restriction in the current version of the ProSet compiler should be removed in the near future.

As explained in Section 5.1, each extension of a branch in the interpretation-tree is formed by appending new entries from $\Omega$, subject to the constraints that (1) each data feature appears at most once on a path through the tree and (2) the data features are used in order (with gaps allowed). The condition in the following `for` loop ensures that these constraints are satisfied:

⟨*Evaluate task for non-wildcard search* 11b⟩≡
```
⟨Check for termination for non-wildcard search 11c⟩
for Entry ∈ hypos | (∀ x ∈ MyPath | (Entry(1) > x(1))) do
  if Consistent (MyPath, Entry) then
    deposit [MyPath with Entry]  at TargetTS end deposit;
  end if;
end for;
```

ProSet's `with` operator adds an element to a set or to the end of a tuple.

Starting from a branch ending with pair $s_\lambda$ (or nothing at the root of the tree), all pairs $s_{\lambda+1} \ldots s_N$ are possible extensions to the branch. Only extensions that satisfy the normal binary constraints are accepted (see the definition of `Consistent` in Appendix C). Extension stops when the termination threshold of matches is reached. The following code checks whether we have enough matches:

⟨*Check for termination for non-wildcard search* 11c⟩≡
```
if #MyPath ≥ Threshold-1 then
  -- We have a satisfactory match except for one data feature:
  TargetTS := RESULT;
else
  -- Deposit a new task for the workers:
  TargetTS := WORK;
end if;
```

`TargetTS` then indicates whether we have a new work task or a new result.

Figure 3 displays the coarse structure of the master-worker program. Arrows indicate access to the tuple spaces. These access patterns are only shown for one of the identical worker processes.

## 5.3 Parallel First-Stop Search Tree Algorithm

This section presents a parallel non-wildcard search tree algorithm which stops the program when the first satisfactory match has been found. This algorithm is quite similar to the sequential non-wildcard search tree algorithm, but the tree is *not* searched depth-first following the leftmost branches first.

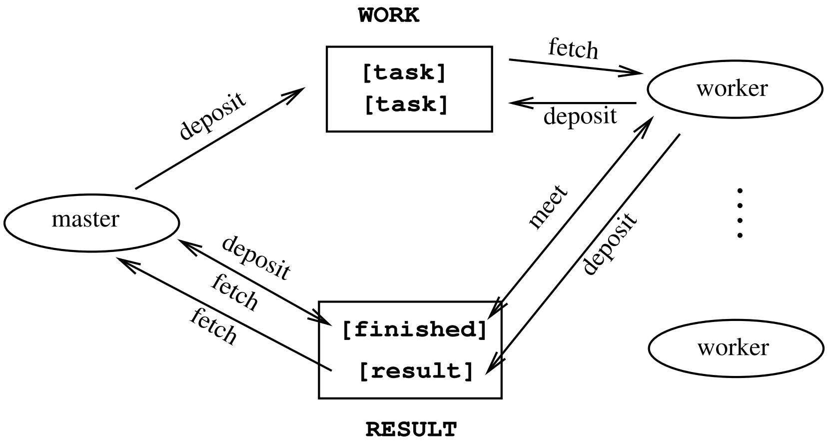


Figure 3: The coarse structure of the master-worker program.

The main program for the parallel non-wildcard first-stop search tree model matching follows:

⟨*First-stop non-wildcard search tree main program* 12a⟩≡
**program** FirstStop;
  ⟨*Tuple space declarations* 8b⟩
  ⟨*Hypotheses from the model invocation* 8c⟩
**begin** -- The master (main program):
  ⟨*Get the number of worker processes* 8d⟩
  ⟨*Get the termination threshold* 9a⟩
  ⟨*Spawn the worker processes* 9b⟩
  ⟨*Deposit the initial task tuples* 9c⟩
  ⟨*Initialize the number of finished workers* 9d⟩
  ⟨*Let the worker processes start working* 9e⟩
  ⟨*Fetch the first result* 12b⟩

  -- The procedure declarations:
  ⟨*Worker procedure for non-wildcard search* 10b⟩
  ⟨*Procedure for reading the hypotheses from the invocation* 32⟩
**end** FirstStop;

The program structure is quite similar to the complete search program (see the chunk indices). The worker procedure is identical. Instead of fetching all satisfactory matches, the master fetches the first satisfactory match from tuple space RESULT and writes the result to standard output, provided that there exists at least one consistent match:

⟨*Fetch the first result* 12b⟩≡
```
fetch ( ? FirstMatch )
                ⇒ put("First match = ", FirstMatch);
  xor ( "Finished Workers", NumWorker )
                ⇒ put("No consistent match found");
   at RESULT
end fetch;
```

Note, that a parallel program terminates when all its sequential processes have terminated. In ProSet, termination of the main program (here the master) terminates the entire application and thus all spawned worker processes.

Synchronization between the master and the workers is achieved when the first satisfactory match has been found. Provided that there exist at least one consistent match, the master need not wait until all tasks are evaluated as is the case with the parallel non-wildcard complete search tree algorithm of Section 5.2.

## 5.4 Evaluation

The parallel non-wildcard complete search tree algorithm provides all satisfactory matches. If we neglect pruning of inconsistent branches, the number of evaluated nodes is proportional to $H^T$ with

$H$ = Number of hypotheses from the model invocation
$T$ = Termination threshold

The time to evaluate these nodes with the sequential algorithm is proportional to $H^T$, whereas the time to evaluate these nodes with the parallel algorithm is proportional to $\frac{H^T}{W}$ with

$W$ = Number of worker processes

because the worker processes evaluate the branches of the tree in parallel. However, the actual amount of parallelism may be restricted by the branching factor of the tree and contention caused by competing access to the tuple spaces. In principle, the situation for the above calculation does not change when considering pruning of inconsistent branches

With the non-wildcard algorithm, the second and third levels of the search tree represent matches that use several non-wildcard pairings. The binary constraints eliminate almost all false pairings quickly [7]. The trade-off is that the branching factor of the non-wildcard tree is $H$ instead of the number of data features as with the standard interpretation-tree algorithm (see Section 3), but the depth of the tree for any false sets of matches is usually very shallow. Therefore, the parallel non-wildcard complete search algorithm allows a high amount of parallelism because of the large branching factor of the tree.

The parallel first-stop search algorithm is quite similar to the sequential non-wildcard search tree algorithm. The tree is not searched depth-first following the leftmost branches first, but in parallel in a non-deterministic order.

For the sequential algorithm, the time to find the first match is highly data dependent. If, for instance, the left-most branch represents a satisfactory match, the sequential algorithm will probably be faster than the parallel algorithm, because the parallel algorithm will probably not follow the left-most branch first. However, consider the example interpretation tree with two paths with satisfactory matches in Figure 4. The satisfactory branches are marked by black circles. Here, the sequential algorithm first evaluates the unsatisfactory left branches before finding the left-most satisfactory match. The parallel algorithm *may* find a satisfactory match earlier, but this is not definite since the evaluation order is non-deterministic. Since the non-wildcard algorithms do not consider any valuation for the data/model feature pairs, nothing can *guide* the workers to follow the most promising branches.

It would be possible to extend the parallel first-stop search algorithm such that it searches the tree depth-first rather than in an arbitrary order. This would improve the probability of finding satisfactory branches earlier, but still the sequential algorithm may be faster.

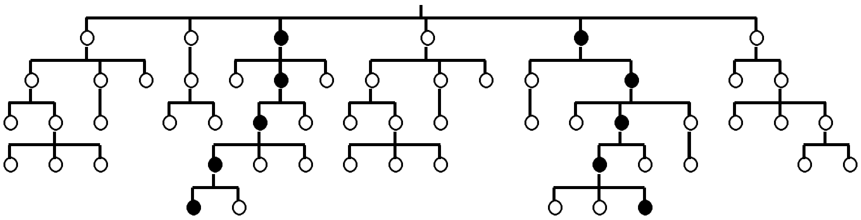

Figure 4: A search tree. The black circles indicate satisfactory matches.

This raises the question whether it pays to parallelize the tree search when we are only interested in obtaining *any* satisfactory match. The situation changes to some extent when we are interested in obtaining a *good* satisfactory match (see the next section).

# 6 Parallel Best Search Tree Algorithms

Using the plausibilities for the model-to-data pairings of the model invocation (see Appendix B), it is possible to define best-search algorithms for interpretation-tree matching. Section 6.1 takes a short look at sequential best-first search tree algorithms. Section 6.2 presents a parallel search tree algorithm which provides the optimal match where each data feature is mapped to a model feature when considering plausibilities for the data/model feature pairs. The sequential best-first search tree algorithm searches for the first plausible solution (usually not the optimal solution). Section 6.3 presents a parallel best-first search tree algorithm.

## 6.1 The Sequential Best-first Search Tree Algorithm

The best-first search tree algorithm [7] assumes that it is possible to evaluate how well sets of model features match sets of data features (these are the plausibilities from the invocation, see Appendix B), and also to estimate the benefit of adding additional feature matches to an existing set of matches. This evaluation can then be used as the basis of a best-first matching algorithm that investigates hypotheses in order of the best estimated evaluation. As any real problem is likely to provide some useful heuristic ordering constraints, the potential for speeding up the matching process is large.

In contrast to the non-wildcard algorithms (Section 5.1), with the best-first algorithms we are interested in both the cost of a path to a solution (i.e. we wish to minimize the time to finding a solution) as well as the quality of the solution. Both algorithms use the same tree structure (see Figure 2 on page 7), but the portion explored may be different.

As with the sequential non-wildcard algorithm, exploration terminates whenever a sufficiently large set of consistent matches is found (termination threshold), or whenever it is impossible to extend the current set of matches to the required number. Refer to [7] for a detailed discussion of sequential best-first search tree algorithms.

### 6.2 Parallel Optimum Search Tree Algorithm

This section presents a parallel search tree algorithm which provides the *optimal* match where a satisfactory number of data features is mapped to model features when considering plausibilities for the data/model feature pairs. The main program follows:

```
⟨Optimum search tree main program 15a⟩≡
  program Optimum;
    ⟨Tuple space declarations 8b⟩
    ⟨Hypotheses from the model invocation 8c⟩
  begin -- The master (main program):
    ⟨Get the number of worker processes 8d⟩
    ⟨Get the termination threshold 9a⟩
    ⟨Spawn the worker processes 9b⟩
    ⟨Deposit the initial task tuples 9c⟩
    ⟨Initialize the result for optimum search 15b⟩
    ⟨Initialize the number of finished workers 9d⟩
    ⟨Let the worker processes start working 9e⟩
    ⟨Wait for the workers to finish 9f⟩
    ⟨Fetch the result for optimum search 15c⟩

    -- The procedure declarations:
    ⟨Worker procedure for optimum search 16a⟩
    ⟨Procedure for reading the hypotheses from the invocation 32⟩
  end Optimum;
```

In addition to putting the initial task tuples into tuple space `WORK`, and initializing a shared counter for the number of finished workers at tuple space `WORK`, the master initializes an empty result set with plausibility 0.0 at tuple space `RESULT`:

```
⟨Initialize the result for optimum search 15b⟩≡
    deposit [ "Optimum", {}, 0.0 ] at RESULT end deposit;
```

After spawning the workers, the master waits until all workers have done their work, and then the master fetches the optimal match from tuple space `RESULT` and prints an appropriate message to standard output:

```
⟨Fetch the result for optimum search 15c⟩≡
     fetch ( "Optimum", ? Match, ? Plausibility ) at RESULT end fetch;
     if Plausibility = 0.0 then
       put("There exists no plausible match!");
     else
       put("Optimal match = ", Match);
       put("Plausibility = ", Plausibility);
     end if;
```

Again, each worker first waits to be enabled and checks whether there are more task tuples in tuple space `WORK`, and terminates when there is no more work to do. Otherwise, the worker again executes in an endless loop:

⟨*Worker procedure for optimum search* 16a⟩≡

```
procedure Worker (i, Threshold);
  ⟨Visibility definitions for auxiliary procedures for optimum search 17d⟩
begin
  ⟨Wait to be enabled 10c⟩
  loop
    ⟨Fetch task 10d⟩
    ⟨Check whether we can prune this subtree 16b⟩
    ⟨Evaluate task for optimum 16d⟩
  end loop;

  ⟨Procedure for consistency check 36a⟩
  ⟨Auxiliary procedure inc2 11a⟩
  ⟨Auxiliary procedure getMyPath 17b⟩
  ⟨Auxiliary procedure getMyPlausibility 17c⟩
  ⟨Auxiliary procedure greater 17e⟩
  ⟨Auxiliary procedure lower-equal 18a⟩
end Worker;
```

Each worker checks whether the plausibility of its current partial match (stored in `MyPath`) is lower than the plausibility of an already known satisfactory match: if so, the worker discards this partial match (according to the bounding rule) and continues to fetch another task tuple. The algorithm assumes that the plausibility evaluation is monotonically decreasing as the path length increases. First the worker computes the plausibility of its own path of matches:

⟨*Check whether we can prune this subtree* 16b⟩≡

```
MyPlausibility := 1.0;
for x ∈ MyPath do
   MyPlausibility *:= x(3);
end for;
```

Then the worker reads the plausibility of an already known optimal satisfactory match (this plausibility is initially equal to 0.0, when no satisfactory match has been found so far) and compares it with the plausibility of its own (not satisfactory) match. If its own plausibility is lower than the plausibility of an already known satisfactory match, the worker continues to fetch another task tuple:

⟨*Check whether we can prune this subtree* 16b⟩+≡

```
meet ( "Optimum", ?, ? OptimumPlausibility ) at RESULT end meet;
if OptimumPlausibility ≥ MyPlausibility then
  continue; -- there exists already a better satisfactory match:
            -- we prune this subtree
end if;
```

If the plausibility of the partial match exceeds that of an already known satisfactory match, the worker checks whether the length of its partial match is already a satisfactory match but one:

⟨*Evaluate task for optimum* 16d⟩≡

```
if #MyPath < Threshold-1
```

If the partial match is not a satisfactory match but one, the worker deposits new task tuples into tuple space `WORK` extended by one pair:

⟨*Evaluate task for optimum* 16d⟩+≡
```
then
  for Entry ∈ hypos | (∀ x ∈ MyPath | (Entry(1) > x(1))) do
    -- Deposit a new task for the workers:
    if Consistent (MyPath, Entry) then
      deposit [MyPath with Entry]  at WORK end deposit;
    end if;
  end for;
```

Again, only extensions that satisfy the normal binary constraints are accepted (see Appendix C).

If the partial match is already a satisfactory match but one, the worker changes the optimal match in tuple space RESULT with a changing meet operation to a satisfactory match based on its own match, provided that these satisfactory matches still have the highest plausibility:

⟨*Evaluate task for optimum* 16d⟩+≡
```
else
  -- We have a satisfactory path but one:
  for Entry ∈ hypos | (∀ x ∈ MyPath | (Entry(1) > x(1))) do
    if Consistent (MyPath, Entry) then
      -- Change the optimum to our path if it is still the best one:
      NextEntry := Entry; -- To make it visible for the auxiliary procedures
      meet ( "Optimum", ? into closure getMyPath($),
                        ? into closure getMyPlausibility($)
                                             | closure greater($) )
       xor ( "Optimum", ?, ?                 | closure lower_equal($) )
        at RESULT
      end meet;
    end if;
  end for;
end if;
```

Again, we need some auxiliary procedures to compute the new values for the optimal match:

⟨*Auxiliary procedure getMyPath* 17b⟩≡
```
procedure getMyPath (x); begin
    return MyPath with NextEntry;
end getMyPath;
```

⟨*Auxiliary procedure getMyPlausibility* 17c⟩≡
```
procedure getMyPlausibility (x); begin
    return MyPlausibility * NextEntry(3);
end getMyPlausibility;
```

These procedures need access to the actual values of the variables MyPath, MyPlausibility and NextEntry. Therefore, these variables are declared to be visible to the local procedures:

⟨*Visibility definitions for auxiliary procedures for optimum search* 17d⟩≡
```
visible MyPath, MyPlausibility, NextEntry;
```

Additionally, we need some auxiliary procedures to check the template conditions:

⟨*Auxiliary procedure greater* 17e⟩≡
```
procedure greater (x); begin
    return (MyPlausibility * NextEntry(3)) > x(3);
end greater;
```

⟨*Auxiliary procedure lower-equal* 18a⟩≡

```
procedure lower_equal (x); begin
    return (MyPlausibility * NextEntry(3)) ≤ x(3);
end lower_equal;
```

### 6.3 Parallel Best-First Search Tree Algorithm

This section presents a parallel best-first search tree algorithm, which terminates at the first satisfactory match. The main program follows:

⟨*Best-first search tree main program* 18b⟩≡

```
program BestFirst;
    ⟨Tuple space declarations 8b⟩
    ⟨Hypotheses from the model invocation 8c⟩
    ⟨Declaration of the float function 21c⟩
begin -- The master (main program):
    ⟨Get the number of worker processes 8d⟩
    ⟨Get the termination threshold 9a⟩
    ⟨Spawn the worker processes 9b⟩
    ⟨Deposit the initial task tuples 9c⟩
    ⟨Initialize the priority queue 18c⟩
    ⟨Initialize the number of finished workers 9d⟩
    ⟨Initialize the number of visited nodes 23a⟩
    ⟨Let the worker processes start working 9e⟩
    ⟨Fetch the result for best-first search 19a⟩
    ⟨Print the number of visited nodes 23c⟩

    -- The procedure declarations:
    ⟨Worker procedure for best-first search 19b⟩
    ⟨Procedure for reading the hypotheses from the invocation 32⟩
end BestFirst;
```

The number of visited nodes is printed for statistical purposes (see Section 6.4).

The central data structure is a distributed priority queue of entries of the following form, sorted by the estimated evaluation of the next potential extension:

$$(S_i = \{pair_{i_1}, pair_{i_2}, \ldots pair_{i_n}\}, g(S_i), m, f(S_i \cup \{pair_m\}))$$

where $S_i$ is a set of $n$ mutually compatible model-to-data pairs, $g(S_i)$ is the *actual* evaluation of $S_i$, $m$ indicates that $pair_m$ is the next extension of $S_i$ to be considered, and $f(S_i \cup \{pair_m\})$ is the *estimated* evaluation of that extension. The priority queue is sorted with larger $f()$ values at the top.

In addition to putting the initial task tuples into tuple space `WORK`, and initializing a shared counter for the number of finished workers, the master initializes the top of the priority queue at tuple space `WORK` with entry $(\{\}, 1.0, 1, A_1)$:

⟨*Initialize the priority queue* 18c⟩≡

```
deposit [ 1, 0, {}, 1.0, 1, hypos(1)(3) ] at WORK end deposit;
```

Each entry of the priority queue is stored as a tuple in `WORK`. The first component indicates the *pointer* to the corresponding entry. The integer `1` indicates the top of the queue. The second component *points* to the next entry. The integer `0` indicates the end of the queue.

Figure 5 on page 22 illustrates the structure of this queue. We shall describe the queue structure later in this section.

The expression `hypos(1)(3)` selects the plausibility for the highest rated hypothesis from the model invocation (this is $A_1$). The hypotheses are initially sorted by the model invocation. See also Appendix B.

After spawning the workers, the master waits until the first worker delivers a match at tuple space `RESULT` and prints an appropriate message to standard output, provided that there exists at least one consistent match:

```
⟨Fetch the result for best-first search 19a⟩≡
    fetch ( "Best", ? Match, ? Plausibility)
                     ⇒ put("Best-first match = ", Match);
                       put("Plausibility = ", Plausibility);
      xor ( "Finished Workers", NumWorker )
                     ⇒ put("No consistent match found");
       at RESULT
    end fetch;
```

Again, each worker first waits to be enabled and then executes in an endless loop:

```
⟨Worker procedure for best-first search 19b⟩≡
    procedure Worker (i, Threshold);
      ⟨Visibility definitions for auxiliary procedures for best-first 20c⟩
      ⟨Declaration of the float function 21c⟩
    begin
      ⟨Wait to be enabled 10c⟩
      loop
        ⟨Pop priority queue top 19c⟩
        ⟨Increment the number of visited nodes 23b⟩
        ⟨Check for termination for best-first search 20d⟩
        ⟨Evaluate task for best-first 20e⟩
      end loop;

      ⟨Procedure for consistency check 36a⟩
      ⟨Procedure for insertion into the priority queue 37a⟩
      ⟨Auxiliary procedure inc2 11a⟩
      ⟨Auxiliary procedure First 20a⟩
      ⟨Auxiliary procedure IsSecond 20b⟩
    end Worker;
```

Each worker then pops the top of the priority queue $(S_i, g(S_i), m, f(S_i \cup \{pair_m\}))$ at tuple space `WORK`:

```
⟨Pop priority queue top 19c⟩≡
     fetch ( 1, ? second, ? S_i, ? gS_i, ? m, ? fS_iPair_m ) at WORK end fetch;
     if second ≠ 0 then
       -- The second entry becomes the first one:
       meet ( ? into closure First($), ?, ?, ?, ?, ? | closure IsSecond($) )
         at WORK
       end meet;
     end if;
```

After popping the top of the priority queue, other worker processes can work in parallel on the tail of the queue, provided that there exists a tail.

Again, we need some auxiliary procedures to compute the integer value `1` and to find the second entry in the queue (this shortcoming in the current version of the ProSet compiler is annoying, but it is just a syntactic restriction):

⟨*Auxiliary procedure First* 20a⟩≡
```
procedure First(x); begin
   return 1;
end First;
```

⟨*Auxiliary procedure IsSecond* 20b⟩≡
```
procedure IsSecond (x); begin
  return x(1)=second;
end IsSecond;
```

Procedure `IsSecond` needs access to the actual value of the variable `second`. Therefore, this variable is declared to be visible to local procedures:

⟨*Visibility definitions for auxiliary procedures for best-first* 20c⟩≡
```
visible second;
```

Extension stops when the termination threshold of matches is reached:

⟨*Check for termination for best-first search* 20d⟩≡
```
if #S_i ≥ Threshold then
  deposit [ "Best", S_i, gS_i ] at RESULT  end deposit;
  return;
end if;
```

or when there are no more hypotheses from the model invocation left:

⟨*Evaluate task for best-first* 20e⟩≡
```
if m+1 > #hypos then
  -- No more hypotheses from the model invocation left:
  if second ≠ 0 then
    continue; -- Evaluate the rest of the priority queue
  else
    -- Nothing more to do:
    meet ( "Finished Workers", ? into closure inc2($) ) at RESULT end meet;
    return;
  end if;
else
  -- We have to evaluate the next hypothesis:
```

Only extensions that satisfy the normal binary constraints (see Appendix C) and the ordering constraints (see Section 5.1) are accepted:

⟨*Evaluate task for best-first* 20e⟩+≡
```
  if Consistent(S_i, hypos(m)) ∧ (∀ x ∈ S_i | (hypos(m)(1) > x(1))) then
     ⟨Generate next descendent of successful extension 21a⟩
  end if;
  ⟨Generate the next descendent of the original popped node 22a⟩
end if;
```

If not rejected by consistency checks, early termination or non-existence of further hypotheses, we generate the next descendent of the successful extension:

$$(S_i \cup \{pair_m\}, g(S_i \cup \{pair_m\}), m+1, f(S_i \cup \{pair_m\} \cup \{pair_{m+1}\}))$$

to be inserted into priority queue.

The algorithm needs two evaluation functions, $f()$ for the estimated new state evaluation and $g()$ for the actual state evaluation. For the above new extension the $g()$ function is set to:

$$g(S_i \cup \{pair_m\}) = g(S_i) * A_m$$

This is one possible state evaluation function. With the sequential best-first algorithms, other state evaluation functions have been investigated [7]. We compute the above state evaluation function as follows:

⟨*Generate next descendent of successful extension* 21a⟩≡
```
g := gS_i * hypos(m)(3);
```

The $f()$ evaluation function is:

$$\begin{aligned} f(S_i \cup \{pair_m\} \cup \{pair_{m+1}\}) &= (size(S_i \cup \{pair_m\}) - 1) + g(S_i \cup \{pair_m\}) * A_{m+1} \\ &= size(S_i) + g(S_i \cup \{pair_m\}) * A_{m+1} \end{aligned}$$

We compute this as follows:

⟨*Generate next descendent of successful extension* 21a⟩+≡
```
 f := float(#S_i) + g * hypos(m+1)(3);
```

The addition of the length of the branch so far gives longer branches higher evaluations to direct the workers to search the tree depth-first. The $g()$ function is monotonically decreasing as the path length increases.

The standard ProSet function `float` converts an integer into a real number. It has to be loaded from ProSet's standard library in the following way:

⟨*Declaration of the float function* 21c⟩≡
```
persistent constant float : "StdLib";
```

The `Insert` function enters the new node into the appropriate priority position, provided that the priority queue contained more than one entry:

⟨*Generate next descendent of successful extension* 21a⟩+≡
```
if second ≠ 0 then
  Insert (1, S_i with hypos(m), g, m+1, f);
else
  next := newat(); -- A new 'distributed pointer'
  deposit [ 1, next, S_i with hypos(m), g, m+1, f ] at WORK end deposit;
end if;
```

If the priority queue contained only the popped entry (the variable `second` indicates this), we directly deposit the new entry as the top of the priority queue. The next entry then obtains the atom `next` as its identity (see below). ProSet's built-in function `newat` returns a new atom.

To update the old state we generate the next descendent of the original popped node:

$$(S_i, g(S_i), m+1, f(S_i \cup \{pair_{m+1}\}))$$

For this new extension the $f()$ evaluation function is:

$$f(S_i \cup \{pair_{m+1}\}) = (size(S_i) - 1) + g(S_i) * A_{m+1}$$

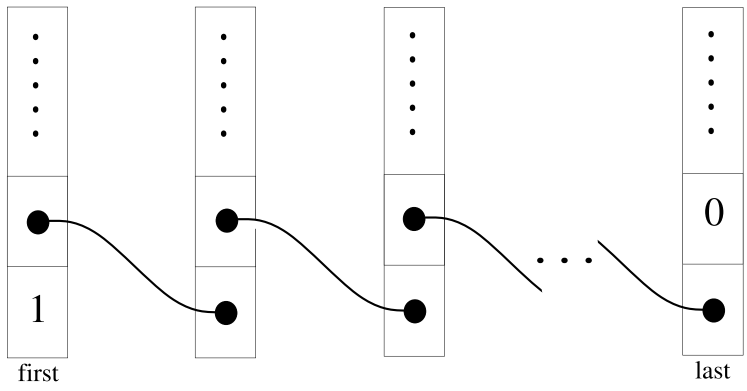


Figure 5: The distributed priority queue for parallel best-first search.
The integer values `1` and `0` are used to indicate the top and the end of the queue, respectively. The intermediate entries are *identified* by the contained atoms. We use black circles to represent the atoms. A link between two atoms means that these two atoms are equal. Note, that the atoms are not the *addresses* of the respective entries, but rather the *identification* of the entries (distributed pointers).

We compute this as follows:

⟨*Generate the next descendent of the original popped node* 22a⟩≡
```
f := float(#S_i - 1) + gS_i * hypos(m+1)(3);
```

and insert the new entry into the priority queue as before:

⟨*Generate the next descendent of the original popped node* 22a⟩+≡
```
if second ≠ 0 then
  Insert ( 1, S_i, gS_i, m+1, f );
else
  if next = om then
     -- There was no descendent of successful extension:
     next := 1;
  end if;
  deposit [ next, 0, S_i, gS_i, m+1, f ] at WORK end deposit;
end if;
```

The priority queue is stored as a *distributed data structure* [18] in tuple space `WORK`. Distributed data structures may be examined and manipulated by multiple processes simultaneously. The individual entries are *linked* together by means of ProSet's atoms. ProSet does not support *pointers* as they are known in Modula, C or similar procedural languages. As mentioned in Section 2.1, atoms are unique with respect to one machine and across machines (they contain the host and process identification, creation time, and an integer counter). Atoms can only be created and compared for equality. We use them as *distributed pointers* which are independent of the processor's memory addresses. The integer values `1` and `0` are used to indicate the top and the end of the queue, respectively. Figure 5 illustrates the structure of this queue. Note, that multiple processes can work independently on different partitions of the queue. The procedure `Insert` for insertion of new entries into the distributed priority queue is presented in Appendix D. A variety of other data structures, such as distributed priority sorted heaps or distributed sorted trees, could be used to implement the priority queue.

### 6.4 Evaluation

The optimum search algorithm is essentially a *branch-and-bound* algorithm [21]. The parallel algorithm searches the same tree as a similar sequential branch-and-bound algorithm would search, but the tree is searched in parallel. The bounding rules apply to parallel search in the same way as they apply to sequential search. Therefore, the parallel optimum search algorithm can be compared to a similar sequential branch-and-bound algorithm in much the same way as the parallel non-wildcard complete search tree algorithm of Section 5.2 can be compared to a sequential non-wildcard algorithm which provides all satisfactory matches. Refer to Section 5.4 for details.

To evaluate the parallel best-first search algorithm, we apply our program to some real data and count the number of visited nodes. The number of visited nodes is initialized by the master:

⟨*Initialize the number of visited nodes* 23a⟩≡
```
deposit [ "Visited Nodes", 0 ] at RESULT end deposit;
```

Each worker increments the shared counter `"Visited Nodes"` in tuple space `RESULT` whenever it evaluates a node:

⟨*Increment the number of visited nodes* 23b⟩≡
```
meet ( "Visited Nodes", ? into closure inc2($) ) at RESULT end meet;
```

After printing the result, the master prints the number of visited nodes to standard output:

⟨*Print the number of visited nodes* 23c⟩≡
```
fetch ( "Visited Nodes", ? NumNodes ) at RESULT end fetch;
put("Number of visited nodes = ", NumNodes);
put("Number of visited nodes per worker = ", float(NumNodes)/float(NumWorker));
```

For testing, the range image of the workpiece displayed in Figure 6 is used. This range image has been processed by the low- and medium-level components of the IMAGINE2 system [8]. The corresponding output of the model invocation is displayed in Figure 7. The experimental results for the parallel best-first search algorithm are displayed in Figures 8 and 9. Figure 8 shows the number of visited nodes in relation to the number of workers and Figure 9 shows the number of visited nodes per worker in relation to the number of workers. $T$ is the termination threshold for satisfactory matches. The zigzag line is due to non-determinism, but the tendency is obvious. The number of visited nodes per worker converges to approximately $\frac{T}{2}$ as the number of workers increases. Therefore, the addition of worker processes increases the search space.

The parallel best-first algorithm appears to be a good compromise between the parallel optimum search algorithm and the sequential best-first algorithm. It is not necessarily much faster than the sequential best-first algorithm, but can produce better results within the same or even a shorter time. The $f()$ function for the estimated new state evaluations directs the workers to search the tree depth-first, which increases the probability of finding a satisfactory match earlier. The workers are *guided* by the plausibilities to follow the most promising branches.

The other main observation to make at this point is: because the sequential variations of interpretation-tree model matching algorithm were presented in a *set-oriented* way [7], it was quite straightforward to implement them and the alternative parallel implementations in ProSet and then compare them, in only a few weeks.

# 7 Implementation of ProSet-Linda

This section briefly discusses the evolving implementation of ProSet-Linda. We implement ProSet-Linda in a somewhat unconventional way: the informal specification is followed by a formal specification, which serves as the basis for a prototype implementation before the production-level implementation is undertaken. Applying formal methods early in the design stage of software systems can increase the designer's productivity by clarifying issues and eliminating errors in the design. A formal development process is more expensive in terms of time and education, but much cheaper in terms of maintenance. There may be bugs, but they are less likely to be at the conceptual level.

The formal specification of the semantics of ProSet-Linda has been presented by means of the formal specification language Object-Z and a prototype for a subset has been implemented from the formal specification with ProSet itself [16, 17]. The prototype allows immediate validation of the specification by execution. It is not possible to check the correspondence between informal requirements and formal specifications formally by verification. The prototype enables us to avoid the large time lag between specification of a system and its validation in the traditional model of software production using the life cycle approach.

In the first C implementation of ProSet-Linda, the SunOS™ 4.1.3 Lightweight Processes Library [27] is used to implement process creation and synchronization. This Lightweight Processes Library only allows quasi-parallel execution on single processor workstations. The C implementation is based on the ProSet prototype implementation (the ProSet compiler translates ProSet into C). In many current operating systems the *lightweight process* or *thread* has emerged as a useful representation of computational activity. Lightweight processes represent multiple threads of control which share the address space of a single heavyweight process. Lightweight processes usually cooperate closely and frequently with each other and are typically used to implement parts of a program which are best executed concurrently. In operating systems like Unix lightweight processes are provided to heavyweight processes by a library which allows the user to execute functions as lightweight processes. Refer to [28, Section 12.1] for an introduction to lightweight processes.

The next implementation was developed for the Meiko™ CS-2 Computing Surface™ at the Edinburgh Parallel Computing Centre. This Computing Surface contains 22 Sparc™ processors connected by an high-speed multi-stage switch network. The CS-2 runs Solaris™ 2.3 (a synonym for SunOS™ 5.3). The re-implementation under Solaris™ 2.3 uses SUN™'s Multi-thread Architecture [24]. This implementation also allows real parallel execution on multi-processor SparcStations™. We use ProSet-Linda on a SparcStation™ 10/512 with two processors. On these multi-processor SparcStations™ the tuple spaces are stored in shared memory.

The CS-2 does not support physically shared memory across processors. On distributed memory architectures, a general problem for implementations of Linda is to provide a map from the virtual shared memory model to physical distributed memory architectures. Therefore, efficient and reliable implementations of Linda on physical distributed memory architectures

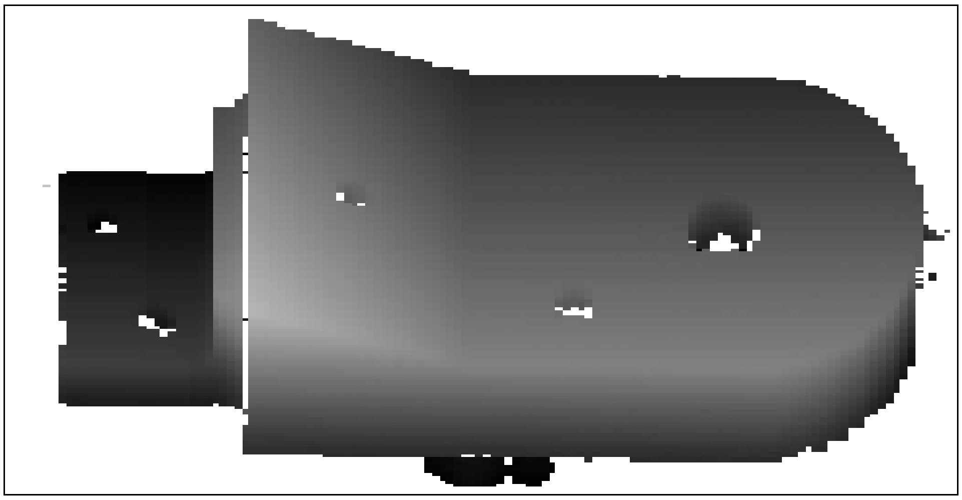

Figure 6: The range image of a workpiece used for testing.

```
Surfhyp plaus 0.824106 context 2 is bae_slope
Surfhyp plaus 0.819578 context 1 is bae_side
Surfhyp plaus 0.819311 context 1 is bae_top
Surfhyp plaus 0.798547 context 2 is bae_hole_side
Surfhyp plaus 0.555499 context 3 is bae_top
Surfhyp plaus 0.553070 context 5 is bae_rect_60x40
Surfhyp plaus 0.553070 context 5 is bae_2hole60x40
Surfhyp plaus 0.447979 context 3 is bae_side
Surfhyp plaus 0.421752 context 6 is bae_rect_10x60
Surfhyp plaus 0.388516 context 2 is bae_side
Surfhyp plaus 0.338191 context 2 is bae_top
Surfhyp plaus 0.309805 context 7 is bae_rect_60x20
Surfhyp plaus 0.305297 context 1 is bae_hole_side
Surfhyp plaus 0.191674 context 5 is bae_prow
Surfhyp plaus 0.134297 context 1 is bae_slope
```

Figure 7: The output of IMAGINE2's model invocation for the range image in Figure 6. See Appendix B for a description of the notation used.

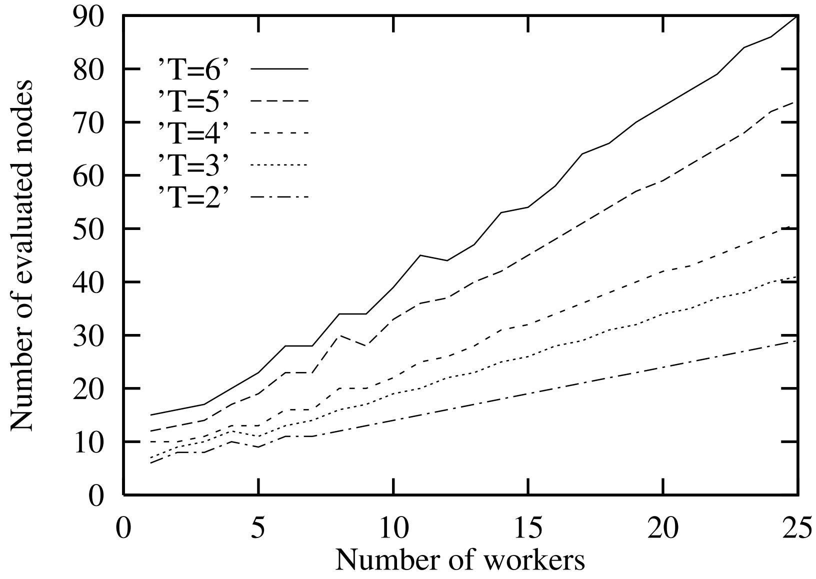


Figure 8: The number of evaluated nodes depending on the number of workers for parallel best-first search.

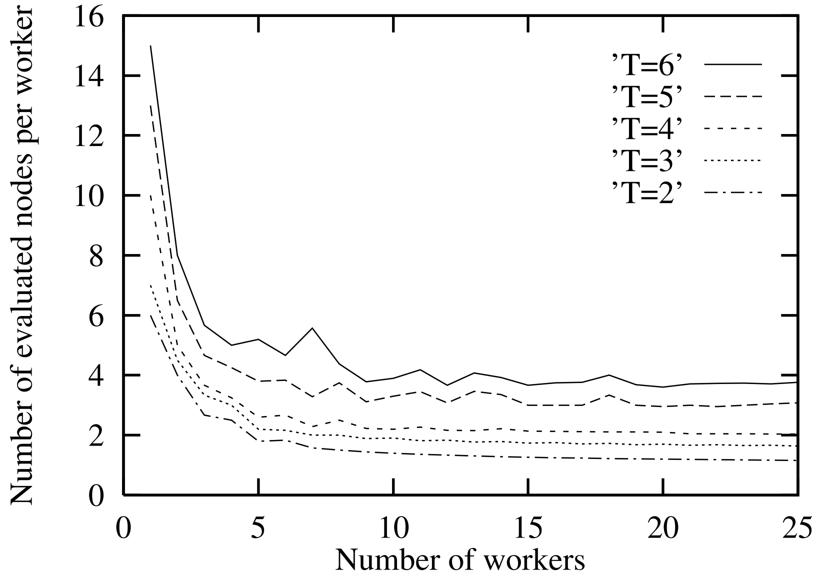


Figure 9: The number of evaluated nodes per worker depending on the number of workers for parallel best-first search.

are in general a great challenge for the implementor. Implementation techniques for physical distributed memory architectures range from ones where the tuple space is replicated on each node to those where each tuple resides on exactly one node. The implementation techniques may be classified as follows:

1. Central store with server process
2. Replication of the entire tuple space at each node
3. Distribution of the tuple space over the net with unique copies of each tuple
4. Mixture of these techniques

A central store may very quickly become both a computational and a communicational bottleneck. Therefore, our first implementation on the CS-2 replicates the tuple spaces on each processor by means of the *Elan widget library* for the CS-2 [22]. This library supports *virtual shared memory* on the operating system level. The virtual shared memory is consistently replicated on each processor belonging to an application program. Updates are performed with atomic broadcast operations. The CS-2 supports the *single-program multiple-data* (SPMD) model for parallel application programs. For ProSet-Linda programs, each process is started in suspended mode except for the main process (executing the main program). Each process contains a server thread to carry out the communication with other processes of the application. New threads can then be created on other processors via appropriate requests to the corresponding server threads (remote thread execution). Accordingly, requests for tuple-space operations are accomplished via appropriate requests to the server threads.

However, a distribution of tuple spaces over the nodes in a parallel system in one form or another is the most promising implementation technique on distributed memory architectures for Linda's tuple spaces. This is due to several reasons. First, memory is saved and second, the overhead for guaranteeing the consistency of the replicated tuple spaces is absent. Furthermore, any Linda implementation that can scale to large machines *must* distribute tuple space, so as to avoid node contention. This distributes the cost of handling tuple operations across all nodes in the system. The remaining problem is *how* to distribute the tuple space. Multiple tuple spaces, as they are supported in ProSet, provide a direct approach for distributing the tuple spaces on a distributed memory architecture. Additionally, the representation of individual tuple spaces can be customized according to their contents and usage. However, a replication of *individual* tuple spaces may also be useful. This would make read operations cheap and write operations more expensive for replicated tuple spaces. Such advanced implementations for ProSet-Linda are the subject for further research.

# 8 Conclusions

We discussed the development of algorithms for parallel interpretation-tree model matching for 3-D computer vision applications with ProSet-Linda. Prototypes for the following algorithms have been developed:

- The parallel complete search tree algorithm.
- The parallel first-stop search tree algorithm.

- The parallel optimum search tree algorithm.
- The parallel best-first search tree algorithm.

The sequential algorithmic variations of interpretation-tree model matching are presented in [7] in a somewhat *set-oriented* way. Therefore, it was quite straightforward to write the computational parts of the parallel variations based on this specification with the set-oriented language ProSet. However, the four presented programs are complete executable prototypes for the developed algorithms. They could be regarded as *executable specifications*.

The evaluation showed that not all algorithmic variations are good candidates for parallelization. An application area for prototyping is to carry out *feasibility studies*. If we had implemented the algorithms directly with a production language, for example C with extensions for message passing, the implementation effort would have been somewhat higher. The implementation of the four prototypes required just a few weeks.

This is what prototyping is about: experimenting with ideas for algorithms and evaluating them. Purely theoretic evaluations are often not possible in practice.

The main contribution of this paper are the presented techniques for parallelization of interpretation-tree model matching and the evaluation of these techniques. It is also a case study for prototyping of parallel algorithms.

## Acknowledgments

This work has been supported by the TRACS program funded by the Human Capital and Mobility program of the European Commission (contract number ERB-CHGE-CT92-0005), and the Universities of Dortmund and Edinburgh.

The authors would like to thank Andrew Fitzgibbon for the help with the IMAGINE2 system, Philippe Fillatreau and Josef Hebenstreit for the discussions on object recognition, and Peter Maccallum and Neil MacDonald for support with the Edinburgh Parallel Computing Centre facilities.

## References


[1] N. Carriero and D. Gelernter. *How to write parallel programs.* MIT Press, 1990.

[2] J. Cocke. The search for performance in scientific processors. *Communications of the ACM*, 31(3):249–253, 1988.

[3] E.-E. Doberkat and D. Fox. *Software Prototyping mit SETL.* Leitfäden und Monographien der Informatik. Teubner-Verlag, 1989.

[4] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, and C. Pahl. ProSet — A Language for Prototyping with Sets. In N. Kanopoulos, editor, *Proc. Third International Workshop on Rapid System Prototyping*, pages 235–248, Research Triangle Park, NC, June 1992. IEEE Computer Society Press.

[5] R.B. Fisher. Model invocation for three dimensional scene understanding. In J. McDermott, editor, *Proc. 10th International Joint Conference on Artificial Intelligence*, pages 805–807. Morgan Kaufmann, 1987.

[6] R.B. Fisher. Non-wildcard matching beats the interpretation tree. In *Proc. British Machine Vision Conference (BMVC92)*, pages 560–569, Leeds, UK, September 1992.

[7] R.B. Fisher. Best-first and ten other variations of the interpretation-tree model matching algorithm. DAI Research Paper No. 717, Dept. of Artificial Intelligence, University of Edinburgh, Edinburgh, UK, September 1994.

[8] R.B. Fisher, A.W. Fitzgibbon, M. Waite, E. Trucco, and M.J.L Orr. Recognition of complex 3-D objects from range data. In S. Impedovo, editor, *Progress in Image Analysis and Image Processing III (Proc. 7th International Conference on Image Analysis and Processing)*, pages 509–606, Monopoli, Bari, Italy, September 1993.

[9] R.B. Fisher, D.K. Naidu, and D. Singhal. Rejection of spurious reflections in structured illumination range finders. In *Proc. 2nd Conference on Optical 3D Measurement Techniques*, Zurich, Switzerland, October 1993.

[10] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[11] W.E.L. Grimson. *Object Recognition By Computer: The Role of Geometric Constraints.* MIT Press, 1990.

[12] W.E.L. Grimson and T. Lozano-Perez. Model-based recognition and localization from sparse range or tactile data. *International Journal of Robotics Research*, 3:3–35, 1984.

[13] R.H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[14] W. Hasselbring. CELIP: A cellular language for image processing. *Parallel Computing*, 14(5):99–109, May 1990.

[15] W. Hasselbring. Prototyping parallel algorithms with ProSet-Linda. In J. Volkert, editor, *Parallel Computation (Proc. Second International ACPC Conference)*, volume 734 of *Lecture Notes in Computer Science*, pages 135–150, Gmunden, Austria, October 1993. Springer-Verlag.

[16] W. Hasselbring. Animation of Object-Z specifications with a set-oriented prototyping language. In J.P. Bowen and J.A. Hall, editors, *Z User Workshop (Proc. Eighth Z User Meeting)*, Workshops in Computing, pages 337–356, Cambridge, UK, June 1994. Springer-Verlag.

[17] W. Hasselbring. *Prototyping Parallel Algorithms in a Set-Oriented Language*. PhD thesis, Department of Computer Science, University of Dortmund, 1994. (Published by Verlag Dr. Kovač, Hamburg).

[18] M.F. Kaashoek, H.E. Bal, and A.S. Tanenbaum. Experience with the distributed data structure paradigm in Linda. In *USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 175–191, Ft. Lauderdale, FL, October 1989.

[19] D.E. Knuth. The `WEB` for structured documentation. Technical Report 980, Stanford Computer Science, Stanford, CA, September 1993.

[20] P. Kruchten, E. Schonberg, and J. Schwartz. Software prototyping using the SETL programming language. *IEEE Software*, 1(4):66–75, October 1984.

[21] E.L. Lawler and D.E. Wood. Branch-and-bound methods: a survey. *Operations Research*, 14(4):699–719, July 1966.

[22] Meiko Limited, Bristol, UK. *Elan Widget Library*, 1993.

[23] D.A. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin. Polaris: A new-generation parallelizing compiler for MPPs. CSRD Report No. 1306, University of Illinois at Urbana-Champaign, Urbana, IL, June 1993.

[24] M.L. Powell, S.R. Kleinman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS Multi-thread Architecture. In *Proc. USENIX Winter '91 Technical Conference*, Dallas, TX, 1991.

[25] N. Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, September 1994.

[26] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets – An Introduction to SETL*. Springer-Verlag, 1986.

[27] Sun Microsystems Inc. *Programming Utilities & Libraries*, 1990.

[28] A.S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[29] E. Trucco and R.B. Fisher. Computing surface-based representations from range images. In *Proc. IEEE International Symposium on Intelligent Control (ISIC-92)*, pages 275–280, Glasgow, UK, August 1992.

# A Literate Programming with `noweb`

This paper has been processed with the literate programming tool `noweb` [25] which is based on Knuth's `WEB` [19]. Literate programmers interleave source code and descriptive text in a single document. Knuth's `WEB` is a tool for writing literate Pascal programs. Ramsey's `noweb` is independent of the target programming languages (for that reason it is prefixed with `no`). Here, we use it for two programming languages, viz. ProSet and C, within one document.

`noweb` is designed around one idea: writing named *chunks* of code in any order, with interleaved documentation. Like `WEB`, and like all literate-programming tools, it can be used to write a program in pieces and to present those pieces in an order that helps to explain the program. A `noweb` file is a sequence of chunks, which may appear in any order. A chunk may contain code or documentation. Code chunks begin with

⟨*Chunk name* 31a⟩≡

on a line by itself. The index 31a is automatically inserted to indicate that this chunk is the first chunk defined on page 31. Documentation chunks are anonymous. Chunks are terminated by the beginning of another chunk, or by end of file. Documentation chunks contain text. If several code chunks have the same name, they are concatenated to produce a single chunk. A code chunk is inserted into another chunk by omitting the ≡ sign:

⟨*Chunk name* 31a⟩

The program `notangle` then extracts the code from the document, and `noweave` outputs a LaTeX file with source code and descriptive text. `noweave` allows one to customize the output of keywords and special symbols within code chunks. ProSet's keyword `forall`, for instance, is displayed within this document as $\forall$. The operator `>=` is displayed as $\geq$. Keywords which are not replaced by mathematical symbols are printed in **bold face**, comments in roman, and the rest in `typewriter` font. `notangle` ignores these translations when extracting the code for subsequent compilation. For an account of literate programming with `noweb` refer to [25].

Program components which are *not* contained within code chunks are displayed in `type writer` font. `noweave` does not customize the output of keywords and special symbols outside code chunks.

An index to objects defined within the code chunks (procedures, variables etc.) can be found at the end of this document. Unfortunately, `noweb` sorts lower case letters behind capital letters.

# B Reading the Output from the Model Invocation

The model invocation uses model and data properties to pair likely model and data features for further consideration. The model invocation process compares the expected and observed properties of features and locally related pairs of features, and assigns a match score based on the closeness of match [5]. It produces a sorted list of consistent model-to-data pairs:

$$pair_i = (model_i, data_i, A_i)$$

where $A_i$ is the compatibility measure (plausibility) of the features $model_i$ and $data_i$. The pair list is initially sorted with larger $A_i$ values at the top.

The corresponding output from the model invocation in the IMAGINE2 system [8] is a list of lines of the form:

```
Surfhyp plaus 0.824106 context 2 is bae_slope
```

`Surfhyp`, `plaus`, `context` and `is` are keywords. The integer `2` is the data feature and the string `bae_slope` is the model feature name. The real number `0.824106` defines the plausibility for this data/model feature pair. The plausibility is a value between $-1.0$ and 1.0, but model invocation only selects pairs for further matching whose plausibility lies between 0.0 and 1.0. The procedure `GetHypos` reads this list from standard input:

⟨*Procedure for reading the hypotheses from the invocation* 32⟩≡

```
@include "gethypos.h"

  procedure GetHypos ();
  begin
    result := []; -- initialize the result tuple
    while NOT_EOF do
      GET_TOKEN;  -- skip 'Surfhyp'
      GET_TOKEN;  -- skip 'plaus'
      p := GET_PLAUSIBILITY;
      GET_TOKEN;  -- skip 'context'
      d :=  GET_DATA;
      GET_TOKEN;  -- skip 'is'
      m :=  GET_MODEL;
      result with:= [d,m,p];
    end while;
    return result;
  end GetHypos;
```

The header file `gethypos.h` defines the `GET_*` and `NOT_EOF` macros (see below).

The above-mentioned input line is translated into the following ProSet tuple representation:

```
[ 2, "bae_slope", 0.824106 ]
```

The output from the invocation is read in by C functions. They are called via ProSet's interface to C:

⟨*Macros for getting the hypos* 33a⟩≡
```
@define  NOT_EOF (c_fct_call check_input() c_string) ≠ "EOF"
@define  GET_TOKEN  c_fct_call get_token()
@define  GET_DATA   c_fct_call get_data() c_integer
@define  GET_MODEL  c_fct_call get_model() c_string
@define  GET_PLAUSIBILITY  c_fct_call get_plausibility() c_real
```

These `@define`s are similar to C's `#define`s. This chunk is tangled into file `gethypos.h`.

The corresponding C profiles:

⟨*C headers for getting the hypos* 33b⟩≡
```
extern char *check_input();
extern void get_token();
extern int get_data();
extern char * get_model();
extern double get_plausibility();
```

This chunk is tangled into file `get.h`.

The C functions:

⟨*C functions for getting the hypos* 33c⟩≡
```
#include <stdio.h>
#include "get.h"

#define MAXBUF 128
static char buffer [MAXBUF];

  ⟨Check for input 33d⟩
  ⟨Get next token 34d⟩
  ⟨Get data feature 34a⟩
  ⟨Get model feature 34b⟩
  ⟨Get plausibility 34c⟩
```

This chunk is tangled into file `get.c`.

Check for end of file:

⟨*Check for input* 33d⟩≡
```
char *check_input()
{
  char ch;

  if ((ch = getc(stdin)) == EOF)
    {
      return "EOF";
    }
  else
    {
      ungetc (ch, stdin);
      return "NOT_EOF";
    }
}
```

Get the index of the next data feature:

⟨*Get data feature* 34a⟩≡
```
int get_data ()
{
  get_token ();
  return atoi (buffer);
}
```

Get the name of the next model feature:

⟨*Get model feature* 34b⟩≡
```
char * get_model ()
{
  get_token ();
  return buffer;
}
```

Get the float value of the next plausibility:

⟨*Get plausibility* 34c⟩≡
```
double get_plausibility ()
{
  double ret;

  get_token ();
  if (sscanf (buffer, "%lf", &ret) == 0)
    {
      fprintf (stderr, "get_plausibility: error from sscanf\n");
      exit (1);
    }
  return ret;
}
```

Store the next token into the `buffer`:

⟨*Get next token* 34d⟩≡
```
void get_token ()
{
  char ch;
  register i;

  ⟨Skip white spaces 34e⟩
  ⟨Read token into buffer 35⟩
}
```

⟨*Skip white spaces* 34e⟩≡
```
  do /* skip white spaces: */
    {
      if ((ch = getchar()) == EOF)
        {
          fprintf (stderr, "get_token: found unexpected EOF\n");
          exit (1);
        }
    }
  while (isspace(ch));
```

⟨*Read token into buffer* 35⟩≡
```
for (i = 0; !isspace(ch); i++)
  {
    if (i >= MAXBUF)
      {
        fprintf (stderr, "get_token: buffer overflow\n");
        exit (1);
      }
    buffer [i] = ch;
    if ((ch = getchar()) == EOF)
      {
        fprintf (stderr, "get_token: found unexpected EOF\n");
        exit (1);
      }
  }
buffer [i] = '\0'; /* terminate string */
```

# C Making Sets of Consistent Model-data Correspondences

The model invocation evaluates *unary* and *binary* geometric constraints for model/data feature pairs (see Appendix B). There exist additional *binary* constraints for combinations of model/data pairs. For example, Grimson and Lozano-Perez [12] provide a set of binary constraints useful for three-dimensional scene analysis, based on *pairwise consistency constraints*, that compare quantities such as relative distance, orientation and direction. Of particular importance is the local nature of the consistency tests, based on the assumption that a few simple, fast tests on partially generated hypotheses will eliminate large numbers of globally inconsistent hypotheses. *Position estimates* can be used to identify features that are visible from the given position, and to eliminate features unlikely to be observed. The model surface patches and the estimated position can be used to determine whether the surface patch is back-facing (not visible).

The procedure `Consistent` checks whether a given pair is consistent with each pair in the path so far:

```
⟨Procedure for consistency check 36a⟩≡
    procedure Consistent (PathSoFar, Pair);
    begin
      for p ∈ PathSoFar do
        if ¬ ConsistentPair (p, Pair) then
          return false;
        end if;
      end for;
      return true;

      ⟨Procedure for consistency check of two pairs 36b⟩
    end Consistent;
```

The procedure `ConsistentPair` checks whether two given pairs are consistent with respect to some geometric constraints (a probabilistic simulation):

```
⟨Procedure for consistency check of two pairs 36b⟩≡
     procedure ConsistentPair (p1, p2);
     begin
        return random(1.0) < 0.8 ;
     end ConsistentPair;
```

For comparison of the different algorithms, we just use a random value to determine the consistency. The emphasis of the present paper is the parallel interpretation-tree search, and not the evaluation of geometric constraints, whose description and implementation would extend to a report by itself.

# D Insertion of New Entries into a Distributed Priority Queue

The procedure `Insert` inserts a new entry into the distributed priority queue of Section 6.3:

⟨*Procedure for insertion into the priority queue* 37a⟩≡

```
procedure Insert (actual, newS_i, newg, newm, newf);
begin
   ⟨Pop top of actual queue 37b⟩

   if newf ≥ fS_iPair_m then
     if next ≠ 0 then
       ⟨Insert before actual entry 37c⟩
     else
       ⟨Insert at end of queue before actual entry 37d⟩
     end if;
   else
     if next ≠ 0 then
       ⟨Insert behind actual entry 38a⟩
     else
       ⟨Insert at end of queue behind actual entry 38b⟩
     end if;
   end if;
end Insert;
```

The workers call `Insert` always with the *pointer* to the top of the priority queue (indicated by the integer `1`). `Insert` may call itself recursively with pointers to tails of the queue. First, the top of the *actual* queue is popped:

⟨*Pop top of actual queue* 37b⟩≡

```
fetch (actual, ? next, ? S_i, ? gS_i, ? m, ? fS_iPair_m )
   at WORK
end fetch;
```

The top of the actual queue is fetched and not just met to guarantee the integrity of the distributed queue. Fetching and returning the entries is a somewhat inefficient implementation of locking. In a production level implementation with, for example, C/C++ with parallel extensions (e.g. C-Linda or message passing) some semaphore mechanism might work more efficiently. In this paper we concentrate on the development of *ideas* for parallel algorithms, not the development of the most efficient implementation at first (this is prototyping).

If the evaluation of the new entry (`newf`) is higher than or equal to the evaluation of the actual entry in the queue (`fS_iPair_m`) and the actual entry is *not* the last entry in the queue (`next` not equal to `0`), then the new entry is inserted in front of the actual entry:

⟨*Insert before actual entry* 37c⟩≡

```
deposit [ actual, next, newS_i, newg, newm, newf ]
     at WORK
end deposit;
Insert (next, S_i, gS_i, m, fS_iPair_m);
```

If the evaluation of the new entry (`newf`) is higher than or equal to the evaluation of the actual entry in the queue (`fS_iPair_m`) and the actual entry is the last entry in the queue, then the new entry is inserted in front of the actual entry at end of the queue:

⟨*Insert at end of queue before actual entry* 37d⟩≡
```
next := newat(); -- We need a new pointer
deposit [ actual, next, newS_i, newg, newm, newf ]
     at WORK
end deposit;
deposit [ next, 0, S_i, gS_i, m, fS_iPair_m ]
     at WORK
end deposit;
```

If the evaluation of the new entry (`newf`) is lower than the evaluation of the actual entry in the queue (`fS_iPair_m`) and the actual entry is *not* the last entry in the queue (`next` not equal to `0`), then the new entry is inserted into the tail behind the actual entry:

⟨*Insert behind actual entry* 38a⟩≡
```
deposit [ actual, next, S_i, gS_i, m, fS_iPair_m ]
     at WORK
end deposit;
Insert (next, newS_i, newg, newm, newf);
```

If the evaluation of the new entry (`newf`) is lower than the evaluation of the actual entry in the queue (`fS_iPair_m`) and the actual entry is the last entry in the queue, then the new entry is inserted at end of queue behind the actual entry:

⟨*Insert at end of queue behind actual entry* 38b⟩≡
```
next := newat();
deposit [ actual, next, S_i, gS_i, m, fS_iPair_m ]
     at WORK
end deposit;
deposit [ next, 0, newS_i, newg, newm, newf ]
     at WORK
end deposit;
```

## Index of Definitions