WOFLAN : a Petri-net-based workflow analyzer

Eindhoven University of Technology
Department of Mathematics and Computing Science

WOFLAN: A Petri-net-based Workflow Analyzer

by

D. Hauschildt, E. Verbeek and W. van der Aalst

97/12

ISSN 0926-4515

editors: prof.dr. R.C. Backhouse
prof.dr. J.C.M. Baeten

Reports are available at:
http://www.win.tue.nl/win/cs

Computing Science Reports 97/12
Eindhoven, August 1997

WOFLAN: A Petri-net-based Workflow Analyzer

Dirk Hauschildt* Eric Verbeek Wil van der Aalst
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513, NL - 5600 MB Eindhoven
{dirk,wsineric,wsinwa}@win.tue.nl

Abstract

Workflow management systems facilitate the everyday operation of business processes by taking care of the logistic control of work. In contrast to traditional information systems, they attempt to support frequent changes of the workflows at hand. Therefore, the need for analysis methods to verify the correctness of workflows is becoming more prominent. In this monograph we present a tool based on Petri nets: *Woflan*. Woflan (WOrkFLow ANalyser) is an analysis tool which can be used to verify the correctness of a workflow procedure. The analysis tool uses state-of-the-art techniques to find potential errors in the definition of a workflow procedure.

*On leave from University of Hamburg, Department of Computer Science, supported by the MATCH project, EC, Human Capital and Mobility, CHRX-CT94-0452

Contents

1 Introduction

Workflow management systems (WFMS) are used for the modeling, analysis, enactment, and coordination of structured business processes by groups of people. Business processes supported by a WFMS are *case-driven*, i.e., tasks are executed for specific cases. Approving loans, processing insurance claims, billing, processing tax declarations, handling traffic violations and mortgaging, are typical case-driven processes which are often supported by a WFMS. These case-driven processes, also called *workflows*, are marked by three dimensions: (1) the process dimension, (2) the resource dimension, and (3) the case dimension (see Figure 1). The process dimension is concerned with the partial ordering of tasks. The tasks which need to be executed are identified and the routing of cases along these tasks is determined. Conditional, sequential, parallel and iterative routing are typical structures specified in the process dimension. Tasks are executed by resources. Resource are human (e.g. employee) and/or non-human (e.g. device, software, hardware). In the resource dimension these resources are classified by identifying roles (resources classes based on functional characteristics) and organizational units (groups, teams or departments). Both the process dimension and the resource dimension are generic, i.e., they are not tailored towards a specific case. The third dimension of a workflow is concerned with individual cases which are executed according to the process definition (first dimension) by the proper resources (second dimension).

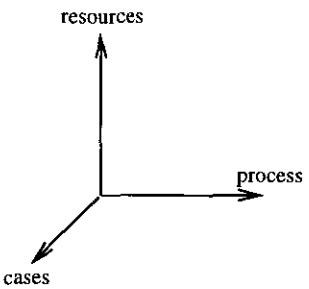

Figure 1: The three dimensions of workflow.

Managing workflows is not a new idea. Workflow control techniques have existed for decades and many management concepts originating from production and logistics are also applicable in a workflow context. However, just recently, commercially available generic WFMS's have become a reality. Although these systems have been applied successfully, contemporary WFMS's do not facilitate advanced analysis methods to determine the correctness of a workflow. Some WFMS's support simulation to validate a workflow definition before it becomes operational. Clearly, there is a need for tools to *verify* the correctness of a workflow definition. This is the reason we developed Woflan.

Woflan focuses on the process dimension (see Figure 1) and assumes a "fair" behavior of the environment (triggering and resources). Woflan also assumes that

a workflow procedure (i.e. the process) is specified in terms of a Petri net. As many researchers have indicated ([EN93, MEM94, WR96]), Petri nets constitute a good starting point for a solid theoretical foundation for workflow management. Unfortunately, most of the more than 250 commercially available WFMS's use a vendor-specific ad-hoc modeling technique to design workflows. In spite of the efforts of the Workflow Management Coalition ([WFM96]) real standards are missing. The absence of formalized standards hinders the development of tool-independent analysis techniques. Fortunately, most WFMS's use a technique which can be translated into a Petri net, because they either use a subclass of Petri nets or use high-level constructs which can be mapped onto them.

Woflan is designed as a WFMS-independent analysis tool. In principle it can interface with many WFMS's. At the moment, Woflan can interface with the WFMS COSA (Software Ley [SL96]) and the BPR tool Protos (Pallas Athena). In the future we hope to extend the set of WFMS's which can interface with Woflan.

This monograph describes the functionality, architecture and implementation of Woflan. The reader is assumed to have some basic knowledge of workflow management and more extensive knowledge of Petri nets. For more detailed information the reader is referred to [WFM96, EN93, Mur89, Aal96a, Aal96c, Aal95, DE95, Aal97, Aal96b, AH97]. This monograph is organized as follows. First, the architecture and input format are discussed. Then a description of the internal data structure and the user interface is given. Finally, the analysis routines are described.

2 Architecture of Woflan

As indicated in the introduction, Woflan (WOrkFLow ANalyser) is a Petri-net-based tool to analyze the correctness of a workflow. The tool is workflow management system independent, i.e. a number of import functions to download workflow-scripts in Woflan (e.g. from COSA and Protos) are provided.

Woflan uses Petri-net-based analysis routines to analyze the workflows at hand. One of the central issues which is analyzed by Woflan is the so-called *soundness property*. A workflow process is sound if, for any case, the process terminates properly, i.e., termination is guaranteed, there are no dangling references, and deadlock and livelock are absent. A formal definition of the soundness property is given in [Aal97, Aal96b]. Clearly, this property is the basic property any workflow should satisfy. In [Aal97] it is shown that standard Petri-net-based analysis techniques to decide liveness and boundedness can be used to verify the soundness property. For many workflows this can be decided in polynomial time. However, Woflan uses a brute force approach by constructing the coverability graph to decide soundness. This turns out to be satisfactory from a practical point of view: even complex workflows have less than 100 tasks (< 200.000 states), this is no problem for Woflan.

Deciding whether the workflow definition is sound is not sufficient. In many cases more requirements need to be satisfied. Moreover, if the workflow definition is not sound, then the user should be guided in detecting the source of the error and support should be given to repair the error. This is the reason Woflan offers a large selection of analysis methods:

- Syntactical checks, e.g., detection of tasks without input or output condition.
- Detection of potential errors by listing suspicious constructs, e.g., constructs violating the free-choice property, AND-split's complemented by OR-join's, OR-split's complemented by AND-join's (well-structuredness), and parts of the net which are not S-coverable.
- Detection of dynamic errors by listing unbounded places, non-safe places, dead transitions and non-live transitions.
- Place and transition invariants. The absence of certain invariants indicates the source of an error.
- Verification of the soundness property.

Woflan has an on-line help-facility which guides the user in using the tool and helps to understand the analysis results.

Woflan consists of three main parts:

- *parser*
  To load a so-called tpn-file representing a workflow process definition.
- *analysis routines*
  A large collection of standard Petri-net analysis routines.
- *user interface*
  To present the results to the end-user.

Figure 2 shows the architecture of WOFLAN.

In addition there will be a module for each WFMS which can interface with Woflan. At the moment there is one such module which can convert COSA script files into a tpn-file. This way Woflan can analyze any workflow process definition constructed by using the COSA network editor (CONE). The same module can be used to import process definitions made with Protos.

A workflow process definition is mapped onto a tpn-file which describes a standard Petri net. Tasks are mapped onto one or more transitions and conditions are mapped onto places. The tpn-file contains information on places and transitions in the net. An example of a tpn-file is:

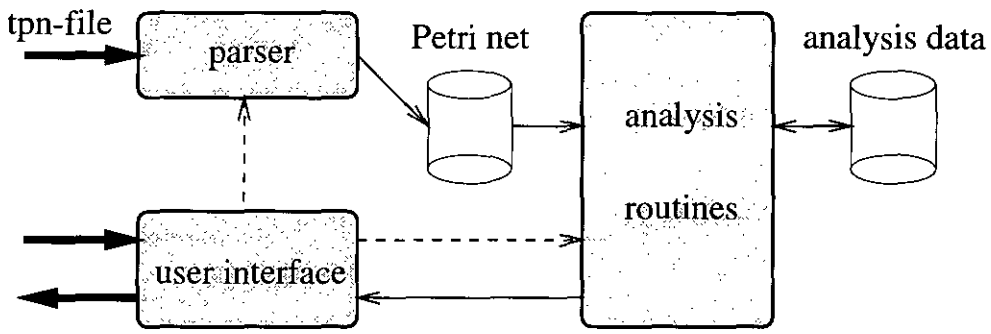

Figure 2: Architecture of WOFLAN.

```
place start init 1;
place c1;
place c2;
place c3;
place c4;
place c5;
place c6;
place c7;
place c8;
place c9;
place c10;
place c11;
place c12;
place c13;
place c14;
place c15;
place c16;
place c17;
place c18;
place c19;
place c20;
place c21;
place c22;
place c23;
place klaar;

trans t1 in start out c1,c7,c15;
trans t2 in c1 out c2;
trans t3 in c2 out c3,c6;
trans t4 in c3 out c4;
trans t5a in c4 out c5;
```

```
trans t5b in c4 out c3;
trans t6 in c5,c6,c14 out c23;
trans t7 in c7 out c8;
trans t8 in c8 out c9,c10;
trans t9 in c6,c9 out c6,c11;
trans t10 in c10 out c12;
trans t11 in c11,c12 out c13;
trans t12 in c13,c19 out c14;
trans t13 in c15 out c16;
trans t14a in c16 out c17;
trans t14b in c16 out c21;
trans t15 in c6,c17 out c6,c18;
trans t16a in c18 out c19;
trans t16b in c18 out c17;
trans t17 in c15 out c20;
trans t18a in c20 out c16;
trans t18b in c20 out c15;
trans t19 in c21 out c22;
trans t20 in c22 out c22;
trans t21 in c22 out c19;
trans t22 in c23 out klaar;
```

Figure 3 shows the example graphically.

3 Data structure

The data structure needed by WOFLAN can be divided into three parts:

1. Classes needed to hold the Petri nets. This structure is build up by WOFLAN's parser and used by its analysis routines.

2. Classes needed to hold necessary analysis data. At the moment this structure mainly holds the net's coverability graph.

3. Classes needed to hold the analysis results. When for instance the place invariants are requested by the user interface, the user interface and the analysis routine concerned have to agree on the way how the routine presents the place invariants to the interface.

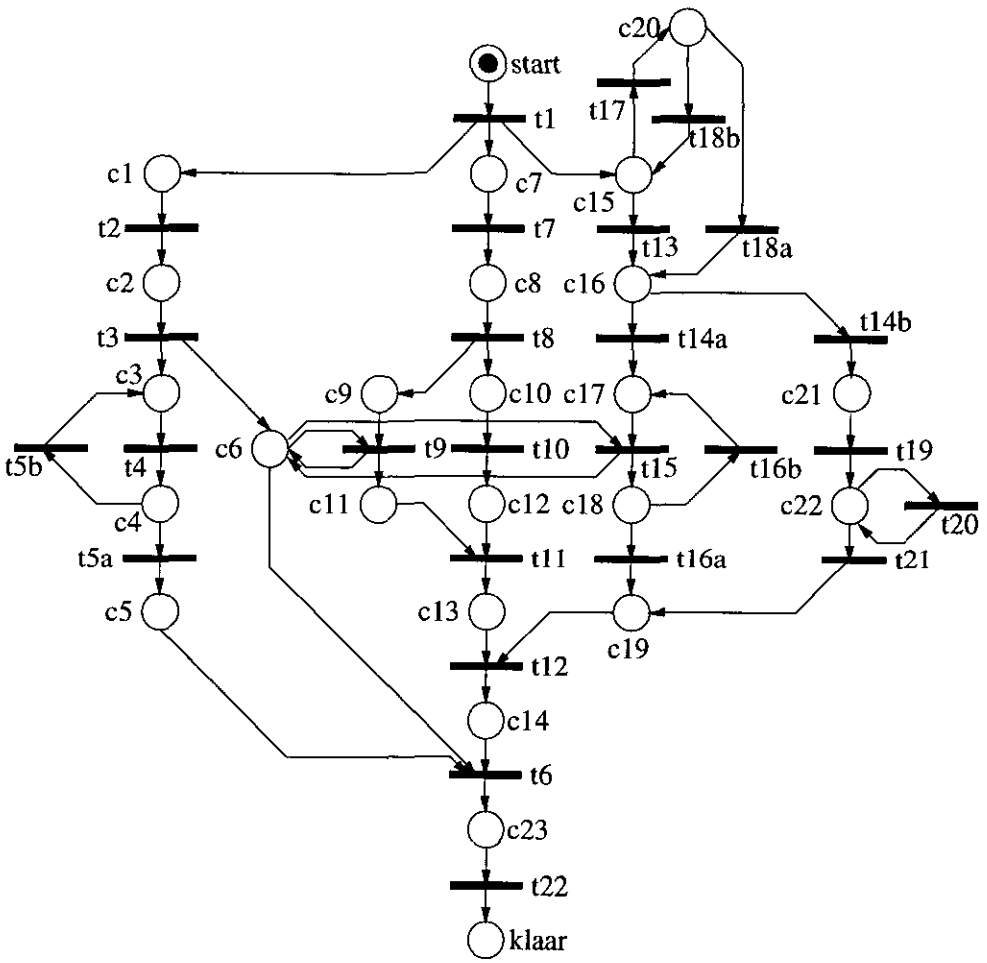

Figure 3: The example.

3.1 Petri net

The object model (OMT) for a Petri net is rather straightforward: a Petri net consists of nodes (which can either be a transition or a place) and connections (which connect one place with one transition and are directed).

Figure 4 shows the object model of WOFLAN's Petri nets.

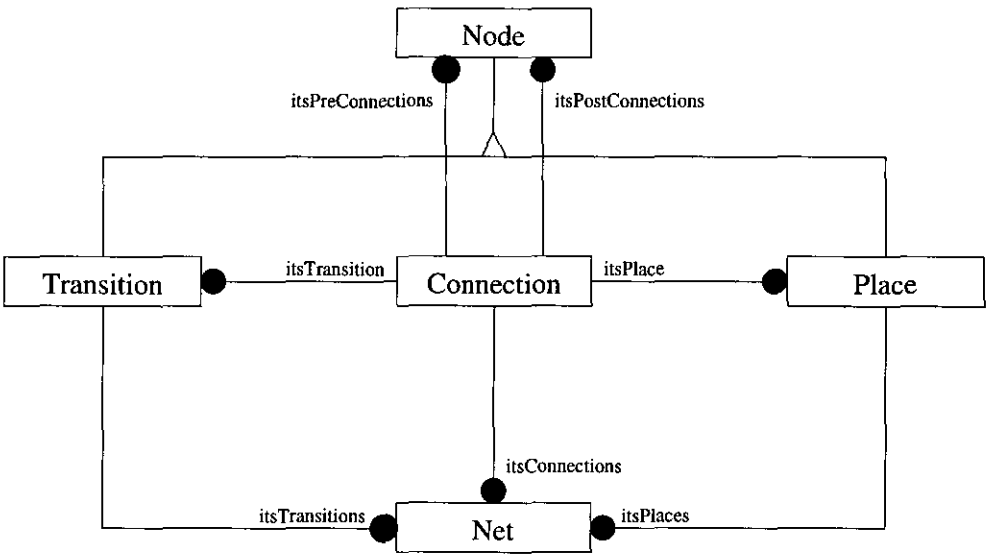

Figure 4: Object model of WOFLAN's Petri nets.

3.2 Analysis data

For the analysis routines the class AnaNet is introduced, which is a derived class of Net. The class AnaNet contains all the analysis routines and the necessary analysis data like the coverability graph.

Figure 5 shows the object model of WOFLAN's analysis data.

3.3 Analysis results

Many analysis routines produce an answer which is built up from the Petri net's nodes, sets and pairs. The routine which returns the place invariants for example, has to produce an answer which is a set of set of weighted places.

The template class `Array` is introduced to hold any set, for instance the class `P_Array` is defined as `Array<Place>`.

Figure 6 shows the object model of WOFLAN's analysis results.

The next sections present the user interface of Woflan and illustrate the analysis methods which can be used to detect and repair errors in a workflow design.

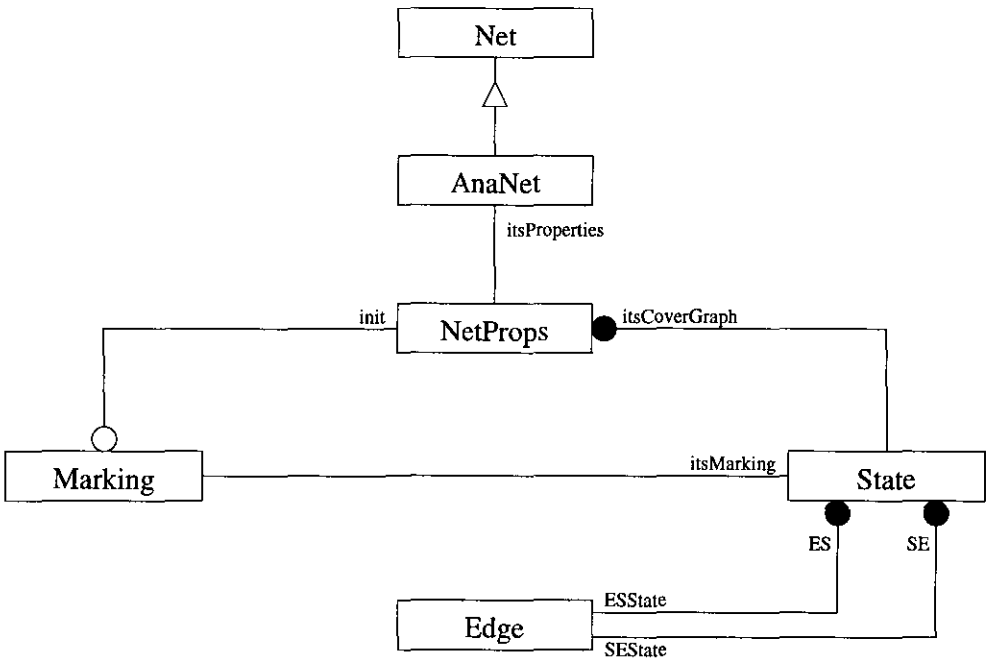

Figure 5: Object model of WOFLAN's analysis data.

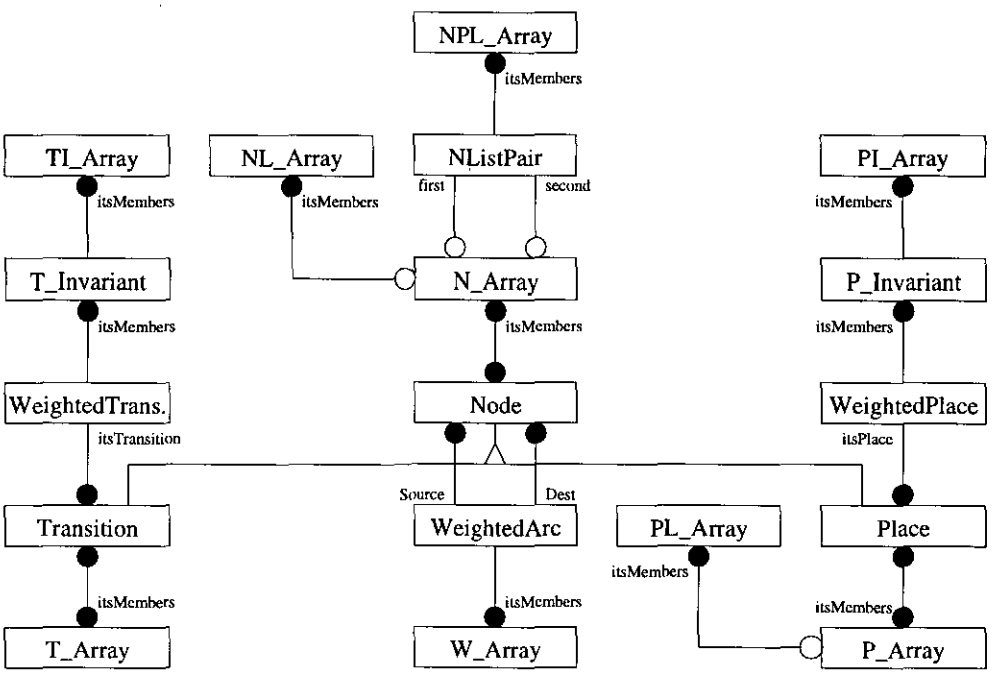

Figure 6: Object model of WOFLAN's analysis results.

4 User interface

The user interface is built using XVT, a GUI builder which supports, among others, both MS Windows as Solaris.

For each Petri net loaded a window is opened containing a button bar (on the lefthandside) and an area where the analysis results are displayed. Using the example which was mentioned earlier we will browse through some of the displays.

The default display is the summary display (see Figure 7). This display shows some basic characteristics of the net, like whether it can be covered by state machines (S-coverable). The coverability graph is not automatically generated, one needs to press a button to do so. Figure 7 shows the summary display after the graph has been generated so it also shows whether the net is bounded etc.

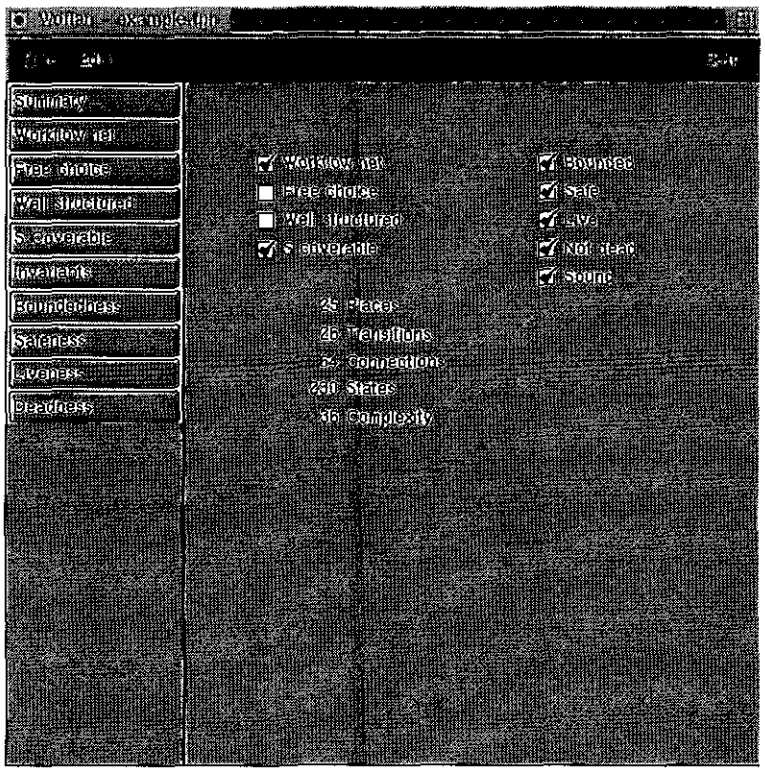

Figure 7: Screendump of WOFLAN's summary display.

The workflow display (Figure 8) shows whether the net is a workflow net (one source place, one sink place, no source nor sink transitions and no (strongly) unconnected nodes) and one can als see for instance which nodes are not connected to any source or sink place. The example net is obviously a workflow net.

The free choice display (Figure 9) shows why the example net is not a free choice net: the cluster `c5 c6 c9 c14 c17 t6 t9 t15` is not free choice. When more than one not free choice cluster is found one can browse through the clusters using the navigation buttons on the lefthandside.

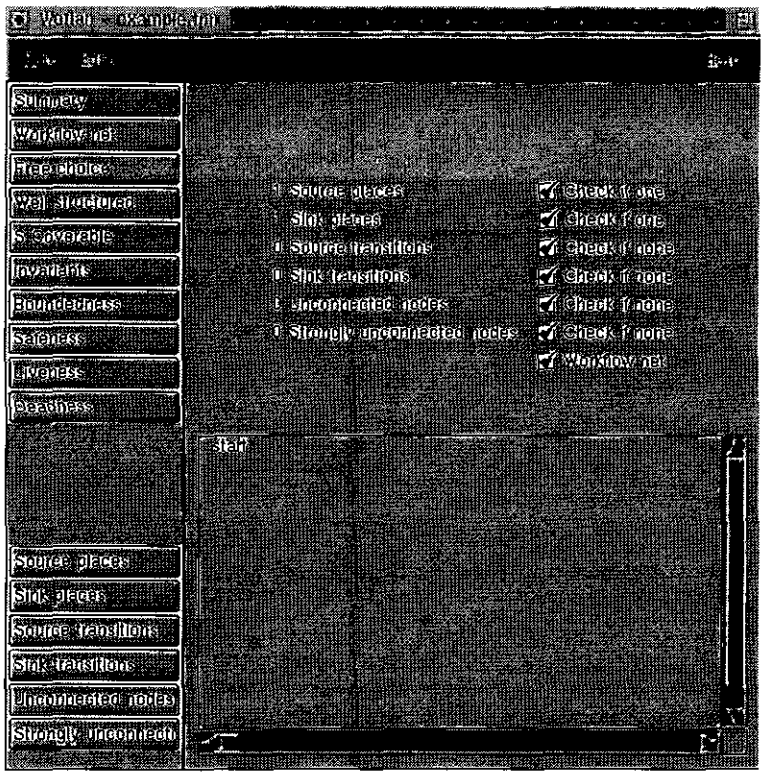

Figure 8: Screendump of WOFLAN's workflow display.

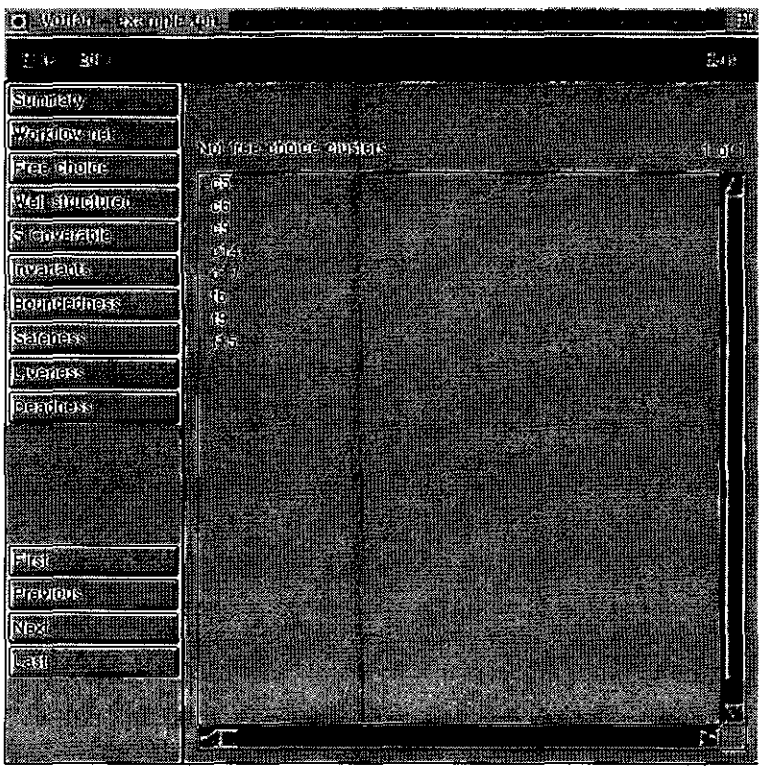

Figure 9: Screendump of WOFLAN's free choice display.

The example net is also not well structured, as is shown by the structure display (Figure 10). Apparently there are nine not well handle pairs (a place-transition pair with multiple paths strongly connecting the second to the first)

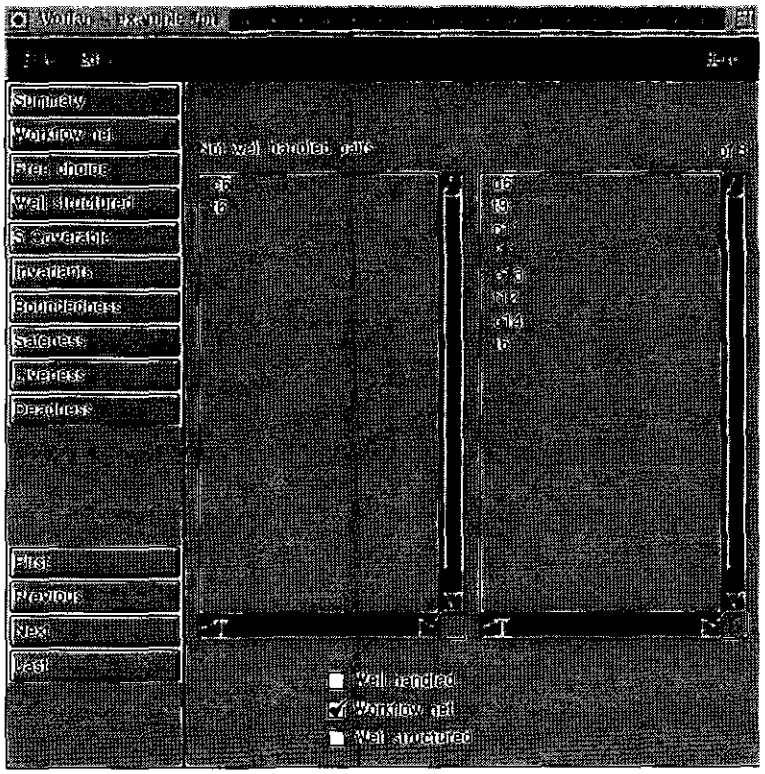

Figure 10: Screendump of WOFLAN's structure display.

The example net however *can* be covered by state machines (five), as is shown by the S-cover display (Figure 11). Nodes which are no part of any S-cover are also listed.

Last the invariant display is shown (Figure 12), displaying (in this case) the semi-positive place invariants found for the example net.

5 Analysis routines

5.1 Introduction

Most of the analysis routines described below can be performed either on the net described in the `Ananet` instance or on an extended version of it, built by introducing an additional transition, called `Extension`, which removes one token from every sink place, i.e., place without output arc (see Section 5.2), and puts one token on every source place, i.e., place without input arc. A 'sound' net, the desirable form of

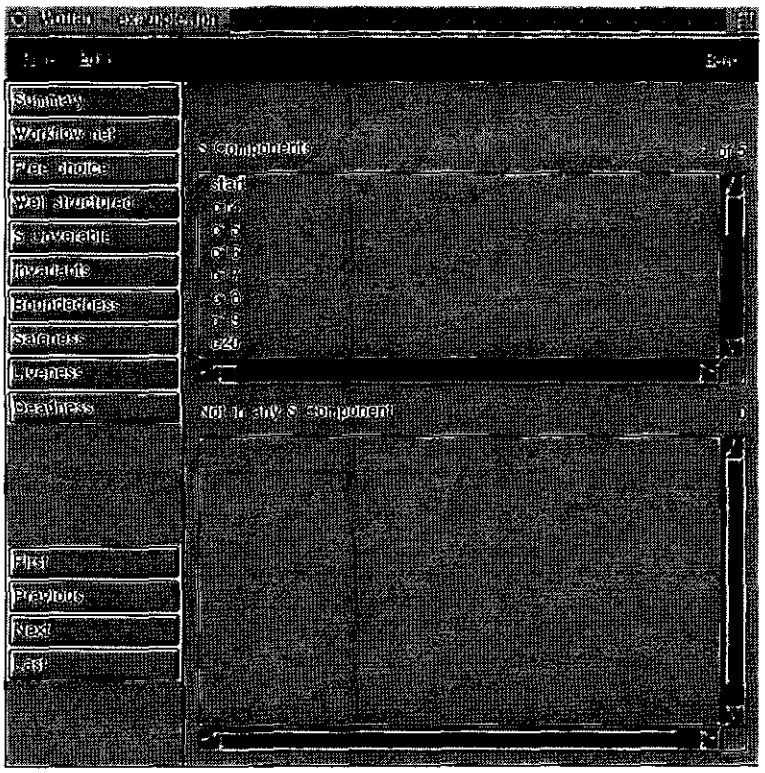

Figure 11: Screendump of WOFLAN's S-cover display.

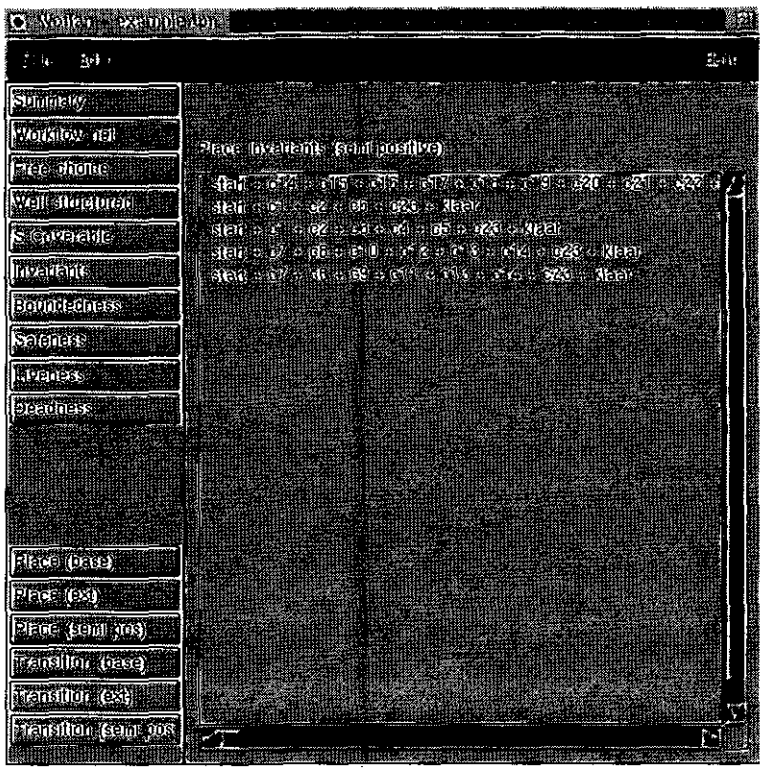

Figure 12: Screendump of WOFLAN's invariants display.

a workflow model, has exactly one source place representing the initial state and one sink place representing the final state. In such a net, the `Extension` transition restarts a completed process again.

There are three groups of analysis routines: the static analysis is mainly considered with graph properties of the net. They are described in Section 5.2. Section 5.3 is devoted to the dynamic analysis which is performed by building up and examining the coverability graph. The invariant analysis which works by solving equation systems derived from the incidence matrix is detailed in Section 5.4. Subsequently, Section 5.5 gives some more technical information about the data structures used to return the results of the analysis, and finally, the appendix describes the equation solver used by the invariant procedures.

5.2 Static analysis

The Petri net is considered as a bipartite graph with the places and transitions as nodes and the net arcs as edges.

5.2.1 The procedures

P_Array `GetSourcePlaces` () const
P_Array `GetSinkPlaces` () const
T_Array `GetSourceTransitions` () const
T_Array `GetSinkTransitions` () const, and

return a list of all places (transitions) without input (output) arc. They are calculated by a simple scan through the list of places (transitions).

5.2.2 The procedure

N_Array `NotConnected` (const Node &handle) const

computes the set of places which are not connected to the given node (parameter `handle`) in the undirected version of the net graph, i.e., where arc directions are ignored. The procedure

N_Array `NotPseudoConnected`
(const P_Array *in = 0, const P_Array *out = 0) const

returns all nodes neither connected to a source place nor to a sink place. Building the extended net explicitly and calling `NotConnected` with the `Extension` transition as parameter `handle` yields the same result. Note that the result of `NotPseudoConnected` is non-empty if and only if any of the connected components contains no source or no sink places (but possibly source or sink transitions). In particular, the procedure returns all nodes if the net is strongly connected itself because it then has no source or sink place at all.

The procedures work by iteratively visiting and marking all reachable nodes and then returning all unseen nodes.

The parameters `in` and `out` are introduced in these and other procedures to avoid repeated computations of the sets of source and sink places. If one of them is non-zero, it should be a pointer to the list of source or sink nodes. If one of them is zero (the default), the set is computed in the procedure.

5.2.3 The procedures

N_Array `NotStronglyConnected` (const Node &handle) const

and

N_Array `NotStronglyPseudoConnected`
(const P_Array *in = 0, const P_Array *out = 0) const

differ from the ones of the previous paragraph only by the fact that they work on the directed net graph. `NotStronglyConnected` returns all nodes which are not reachable from the node parameter `handle` or from which this node is not reachable.

A node belongs to the result of `NotStronglyPseudoConnected` if it is not reachable from any source place or no sink node can be reached from it.

The procedures perform two sweeps through the net to mark all forward or backward reachable nodes and then return all nodes not contained in the intersection of the two node sets computed this way.

5.2.4 A net is called well-founded if it has exactly one source place, exactly one sink place, and it is strongly connected. The procedure

int `IsWellFormedNet` (const P_Array *in = 0, const P_Array *out = 0) const

tests for this property. There is no extended version of it since a net extension cannot be well-founded by definition.

5.2.5 A cluster of a net N is a connected component of the net N' obtained from N by deleting all transition-to-place edges. Let P be the set of places in a cluster C and T contain its transitions. C is called free-choice if all transitions in T have the same set of input places (namely P), and consequently all places in P have T as their set of output transitions.

The procedure

NL_Array `NotFreeChoiceClusters` () const

calculates the set of clusters not enjoying this property, and

int `IsFreeChoice () const`

answers the question of whether the net contains such a cluster.

The procedures work by scanning through the list of transitions. If a transition t does not belong to a cluster visited before, the cluster containing t is calculated by iteratively traversing all place-to-transition edges (directly or indirectly) connected

to t. During this calculation, the free-choice property is tested by comparing the preset of every transition in the cluster with the one of t. If any differences are encountered, the cluster is output as non-free-choice.

There may be clusters not detected by the procedure, but they consist of a single sink place and have the free-choice property.

5.2.6 The procedure

W_Array `MultipleArcs` () const

computes all ordered node pairs (n_1, n_2) for which there exists more than one edge from n_1 to n_2.

int `HasMultipleArcs` () const

returns true if such a pair exists. Impure nets, e.g., nets with node pairs (n_1, n_2) and arcs both from n_1 to n_2 and from n_2 to n_1 are not detected.

The procedure scans through all edges and conducts some bookkeeping about the sets of connected node pairs. If an edge between two already connected nodes is encountered, the pair is put into the return set.

5.2.7 A pair (n_1, n_2) of nodes consisting of one place and one transition is called not-well-handled if there are two node-disjoint paths (except for the start and end node) leading from n_1 to n_2.

Two paths starting (ending) in a place usually describe the begin (end) of two alternative subprocesses while different edges starting (ending) in a transition correspond to parallel subprocesses. Not-well-handled pairs normally derive from a mismatch of these two concepts are therefore a likely indicator for an error in the design of the model.

The procedures

NPL_Array `NotWellHandledPairs () const`

and

NPL_Array `NotWellStructuredPairs`
(const P_Array *in = 0, const P_Array *out = 0) const

compute the set of not-well-handled pairs in the net itself, and in its extension, respectively.

The mere question of whether such pairs exist are answered by the procedures

int `IsWellHandled` () const

and

int `IsWellStructured` (const P_Array *in = 0, const P_Array *out = 0) const.

To test for disjoint paths, we apply the max-flow min-cut technique as described e.g. in [Eve79]. Its problem is given by a directed graph with a designated start

node s, a designated final node f, and a maximal edge capacity for every edge. The method allows to compute some edge weights lying between 0 and the corresponding capacities for which the flow (number of units transported simultaneously) from s to f is maxiaml.

To apply the algoritm to our problem, we have to (conceptually) replace every node n by a node n' inheriting all input edges of n, a node n'' inheriting all output arcs, and a new edge from n' to n''. If we set the capacity of all new edges to one, the maximal flow between s'' and f' is equal to the number of disjoint paths between s and f in the original graph. The capacities ensure that every node is visited at most once.

Our algorithm is a simplification of the algorithm proposed in [Eve79] which makes use of the fact that we just want to find out whether or not the flow exceeds one.

5.3 Dynamic analysis

The procedures described in this section have to to with the coverability graph of the net. They either build it up or analyse its properties. For nets with a finite reachability set, this graph is identical to its reachability graph.

Except for `DeleteCoverabilityGraph` and `sound`, all procedures carry two parameters `init`, describing the initial marking of the net (or the initial state of the system), and `extend` which indicates whether the graph should be built for the ordinary net or its extension. `init` is of the type `LMarking`, which is a synonyme for `*unsigned`. The length of the array is implicitly given by the number of places in the net.

Since building up the graph can be time-consuming, it is not deleted after a call of an analysis procedure, but preserved for subsequent calls. At the begin of every call it is checked whether a graph already exists and, if so, whether it was built up with the same initial marking as given in the procedure call. If this is not the case, the graph is (re-)built.

A procedure called with `extend` = 0 also works with an extended coverability graph and a call with `extend` = 1 may lead to an extension (rather than a rebuild) of a possibly existing graph for the ordinary net. Nevertheless, building an ordinary graph and later extending it is slower than building up the extended graph in the first place since some auxiliary data structures have to be recomputed. We now describe the procedures in detail.

5.3.1 The procedure

unsigned `GenerateCoverabilityGraph`
(const LMarking init, int extend = 0, int preserve = 0) const

builds up the graph unconditionally even if a graph with the same initial marking exists. It returns the number of states in the graph. This may be useful if the net has changed.

Setting an optional parameter `preserve` prevents the rebuild if the initial marking is still the same. This way the procedure can be used to query the size of the graph.

There exist different versions of coverability graphs and trees in the literature. One distinction is made by the question of whether two nodes with the same marking, but reached on different firing sequences should be joined into one and when and how often in the process of building the node these equality test are performed.

Another choice has to be made for the question of when a finite number of tokens on a place has to be replaced by an 'ω'-symbol (meaning that the number can get arbitrary high) to avoid the construction of an infinite graph.

Our procedure always joins two nodes with identical markings. A distinction is made between 'tree edges' leading to a new node and 'other edges' connecting already existing nodes. Whenever the procedure finds an enabled transition in the marking of a certain node n, it performs the following steps.

(i) It first builds the successor marking.

(ii) It then checks for existing identical markings in the graph.

(iii) If not successful, it checks for smaller or equal markings on the unique path from the root to n consisting of tree edges. It sets the number of tokens to ω on all places for which a smaller or equal marking was found which differs for this place. Even if the marking has changed this way, it does not restart the search process to find some markings smaller or equal to the changed marking.

(iv) If the last step resulted in a change of the marking, another equality check is performed.

If one of the equality checks was successful, an 'other edge' is inserted into the graph, otherwise a new node is created and connected by a 'tree edge'.

5.3.2 The procedure

void `DeleteCoverabilityGraph` () const

destroys the graph. It should be called on changes of the net or simply to recover the memory space used by the graph.

5.3.3 The procedure

P_Array `Not_k_Bounded` (const LMarking init, unsigned k, int extend = 0) const

carries an additional parameter k and computes the set of places for which a reachable marking exists on which the place holds more than k tokens. To just find out whether such places exist, one can call the procedure

int `Is_k_Bounded` (const LMarking init, unsigned k, int extend = 0) const.

Both procedures work by traversing the whole coverability graph along the tree edges. They stop as soon as all nodes are found to be not k-bounded, or, in case of `Is_k_Bounded`, the first not k-bounded node has been found.

5.3.4 The procedures

P_Array `NotBounded` (const LMarking init, int extend = 0) const and
int `IsBounded` (const LMarking init, int extend = 0) const

are interfaces to the procedures described in the previous paragraph. They can be used to test for unbounded places.

Similarly

P_Array `NotSave` (const LMarking init, int extend = 0) const and
int `IsSave` (const LMarking init, int extend = 0) const

just call `Not_k_Bounded` and `Is_k_Bounded` with $k = 1$.

5.3.5 The procedure

T_Array `Dead` (const LMarking init, int extend = 0) const

computes the set of transitions which do not occur in any firing sequence starting at the initial marking of the net.

int `HasDead` (const LMarking init, int extend = 0) const

just investigates the question of whether such transitions exist.

Similar to `Not_k_Bounded` and `Is_k_Bounded`, these functions traverse the graph and maintain a set of transitions not encountered so far. They stop as soon as this set is empty or the whole graph has been traversed.

5.3.6 The procedure

T_Array `NotLive` (const LMarking init, int extend = 0) const

is actually a misnomer. A transition is included into the output set if there exist a node in the coverability graph from which no arc labelled by this transition can be reached. Therefore $\{t\}$-continuality of the net (existence of an infinite firing sequence in which t occurs infinitely often) is a necessary but not sufficient condition for t not to be included in the output set of this procedure. The latter property, in turn, is necessary but not sufficient for liveness. But for bounded nets, the procedure computes exactly the set of non-live transitions.

The procedure

int `IsLive` (const LMarking init, int extend = 0) const

tell us whether the set returned by NotLive is empty.

Both procedures calculate all strongly connected components of the coverability graph with the additional property that no arc from inside the component is leading to an outside node. More exactly, they compute the set of root nodes of these components.

This computation is done in linear time by a depth first search through the graph and actually simpler than calculating all strongly connected components. Once a component is found, an interior loop is started by a call to `Dead`. It computes the set of transitions not occurring in the component. The union of these sets is returned as not live. Since also `Dead` works in linear time and every node belongs to only one component, and therefore is visited in at most one call of `Dead`, the overall complexity remains linear to the size of the graph.

As in the previous procedures, the search is stopped when all transitions are found to be non-live and a call to `Dead` proceeds only as long as it can possibly decover a new non-live transition.

5.3.7 A net is called sound if it is well-founded and the reachability set from the initial marking consisting of exactly one token lying on the only source place has four additional properties.

(a) There is only one reachable marking containing tokens on the only sink place.
(b) This marking, called the final marking, contains one token on the sink place and all other places are empty.
(c) The final marking is reachable from all other reachable markings.
(d) There are no dead transitions.

One can show that for well-founded nets, these four conditions are equivalent to the extension of this net being live and bounded.

The algorithm to check for soundness follows the above definition closely. If the net is well-founded at all, the coverability graph from the initial marking defined above is built and then up to four traversals through the graph are started to test for the properties above. Properties (a) and (b) can be verified simultaneously but two sweeps are necessary for property (c).

The initial marking is given implicitly in the definition of soundness. Moreover, since an extended net is never well-founded, and therefore also never sound, `extend` can be assumed to be zero. Thus both of the usual parameters `init` and `extend` are obsolete and omitted.

5.3.8 Let an equivalence relation between the places be defined by the fact that they contain the same number of tokens in every reachable marking. Then we can partition the set of places into a set of equivalence classes. The procedure

PL_Array `EquivalentPlaces` (const LMarking init, int extend = 0) const

lists all such equivalence classes containing more than one place.

int `HasEquivalentPlaces` (const LMarking init, int extend = 0) const

can be called to find out whether there are any equivalent places at all.

For every reachable marking, every equivalence class calculated so far is splitted into a set of places containing as many tokens as the first place in the class and the set of other places. The latter one is appended to the list of equivalence classes and might be splitted again during the investigation of the actual marking. Sets with less than two places are discarded during this computation. The computation is stopped as soon as the set of remaining equivalence classes is empty.

5.3.9 Let a pair of places be called exclusive if there is no reachable marking in which both of the places contain at least one token. The set of such pairs is computed in the procedure

PL_Array `ExclusivePlacePairs` (const LMarking init, int extend = 0) const.

There is also a procedure

Int `HasExclusivePlacePairs` (const LMarking init, int extend = 0) const

which just answers the question of whether there are any such pairs.

Again, the procedures sweep through the graph. They maintain a list of places which are still exclusive to other places and keeps a `BitSet` for every place in the list which contain the set of places exclusive to it so far.

In the treatment of a certain marking, the set of non-empty places in this marking is subtracted from any `BitSet` kept for a marked place in the list. The place is removed from the list as soon as the `BitSet` becomes empty this way. The whole search is stopped when the list itself becomes empty.

5.4 Invariant Analysis

Solutions x of $C \cdot x = 0$ where C is the incidence matrix of the net and x is a vector containing a weight factor for every place of the net are called P-invariants. They have the property that the scalar products $x \cdot m$ and $x \cdot m'$ are identical for every pair m, m' of markings where m' is reachable from m.

Similarly, solutions x of $C^T \cdot x = 0$ are called T-invariants. If the Parikh image of a firing sequence is a T-invariant, the marking after the occurrence of such a sequence is the same as before.

The procedures described in this section compute different representations of these sets of invariants. Like in the last section, they all contain a parameter `extend` which determines whether invariants of the original or of the extended net are to be computed.

5.4.1 If one allows negative weight factors to occur in the invariants, the invariant set forms a vector space. The procedures

PI_Array	`PlaceInvariantsBase`	(int extend = 0) const
TI_Array	`TransitionInvariantsBase`	(int extend = 0) const

compute a base $B \subseteq \mathbb{Z}^n$ of it, where n is the number of places. Every integral invariant has a representation as a linear combination of B with integral coefficients and every rational invariant can be constructed with rational coefficients.

B is computed by calling the equation solver contained in the program `gold` described in the appendix.

5.4.2 The procedures

PI_Array	`PlaceInvariantsSemiPos`	(int extend = 0) const
TI_Array	`TransitionInvariantsSemiPos`	(int extend = 0) const

deal with non-negative solutions. They compute a set B containing the minimal elements of the set of non-negative non-zero solutions of the equation system. Every invariant $x \subseteq \mathbb{N}^n$ can be represented as a linear combination of B with natural numbers as coefficients.

Both procedures incorporate the program `gold` with the parameters '`/n`' and '`/m`'.

5.4.3 The procedures

PI_Array	`PlaceInvariantsExt`	(int extend = 0) const
TI_Array	`TransitionInvariantsExt`	(int extend = 0) const

attempt to calculate all 'intuitively useful' basic invariants in $\mathbb{Z}^n$ by incorporating an adaption of the inequality solver.

Given a system $A \cdot x \geq 0$ of equations and inequalities with n variables, these procedures simultaneously solve 2^n inequality systems. They all are obtained from the original system by adding either the inequality $x_i \geq 0$ or $-x_i \geq 0$ for every variable x_i.

Let the support of a vector be the set of its non-zero components. Then the procedure computes a set B of all non-zero solutions x of $A \cdot x \geq 0$ with minimal support, i.e., the support of no other non-zero solution is a strict subset of the support of an $x \in B$.

Consequently, every invariant in $\mathbb{Z}^n$ has a representation as a linear combination of B with natural numbers as coefficients. Empirical observations have shown that these procedures are fast enough for P-invariants, but often turn out to be too slow if T-invariants are to be computed.

5.4.4 For a node n (a place or a transition) n of a net let ${}^\bullet n$ be the set of nodes n' for which there exists an edge from n' to n and $n^\bullet$ be the set of nodes n' with an edge from n to n'.

An S-component C of a net N without multiple arcs is a non-empty, strongly connected subnet of N with two additional properties.

(a) For every place p in C, both ${}^\bullet p$ and $p^\bullet$ belong to C as well.
(b) For every transition t in C, we have $|C \cap {}^\bullet t| = 1 = |C \cap t^\bullet|$.

One can show that any S-component consists of the support of a minimal support P-invariant and the transitions adjacent to these places. But not every such P-invariant defines an S-component.

If the net has no multiple arcs, any P-invariant defining an S-component has no other weight factors than 0 or 1. The S-components are computed by the procedure

NL_Array `S_Components` (int extend = 0) const.

The calculation is done in two steps. First the P-invariants are computed by a call to the program `gold` with the parameters '`/r`' and '`/n`'. In the second step, every invariant is augmented by adding all transitions connected to a place belonging to its support and a check for Condition (b) above is performed.

There are two more procedures dealing with S-components.

N_Array `NotIn_S_Components` (int extend = 0) const

joins all S-components as defined above into one set and then returns the complement of it and

int `S_Cover` (int extend = 0) const

returns true iff the set computed by `NotIn_S_Components` does not contain any place (but possibly some isolated transitions).

5.5 Data Structures

The analysis routines either return some boolean information in a variable of type `int` (coded as usual: 0 for false and 1 for true) or they use a data structure called `Array<T>` to return a list of pointers to objects of a type `T`.

The arrays can be accessed by two functions

int `Array<T>::length` () const

which returns the number of elements in the array and

T & `Array<T>::operator []` (unsigned) const

to address some element. If `a` is an array and $0 \leq i \leq$ `a.length ()`, then `a[i]` returns the i-th element.

There is also a function

T * `Array⟨T⟩::Get` (unsigned i) const

which returns a pointer rather than a reference to the element. Moreover, it checks for the validity of its argument and returns 0 if it is not smaller than `a.length ()`.

The arrays are returned by value, because this is the only way to create array objects in the scope of the calling procedure which are later automatically deleted without requiring an explicit `delete`-statement.

`C++` uses the `Array⟨T⟩::Array` (Array &) constructor to initialize procedure arguments and return values if they are defined as value parameters. For the arrays defined here, these operators are defined in such a way that they copy the 12-byte header, but not the actual pointer array. The same holds for the (rarely used) assignment operator.

This saves a lot of copying (especially for nested arrays) but it implies that arrays given away as value parameters cannot be any more accessed in the calling procedure. If copying is really necessary, it is done through the `Array⟨T⟩::Array` (const Array *) constructor.

The intended way of using a procedure `proc` returning an array is either by

`Array⟨T⟩` a = `proc` (...);

in which case the array is deleted at the end of the scope of the variable a (or it is further returned by value as a function result) or by

`Array⟨T⟩` *a = new `Array⟨T⟩` (`proc` (...));

eventually followed by an explicit

`delete` a;

There is a subtype of `Array⟨T⟩` called `DelArray⟨T⟩` which only differs by the fact that upon deletion of a `DelArray⟨T⟩` not only the array of pointers to instances of T is discarded, but also the instances themselves.

These two data types allow us to build up the data structures in such a way that upon deletion of the return value of an analysis procedure all objects created in the procedure are deleted as well.

In the simplest case, a procedure returns a list of net elements in one of the data types

`P_Array` = `Array⟨Place⟩`,
`T_Array` = `Array⟨Transition⟩`, and
`N_Array` = `Array⟨Node⟩`.

If the list elements are created in the procedure, the list has the form `DelArray⟨T⟩` as in

W_Array = DelArray⟨WeightedArc⟩,
P_Invariant = DelArray⟨WeightedPlace⟩, and
T_Invariant = DelArray⟨WeightedTransition⟩.

A WeightedArc is used in the file MultipleArcs.h and has three components

const Node *Source, const Node *Dest, and unsigned Weight.

The last components holds the number of arcs from *source to *dest.

Similarly, the types P_Invariant (T_Invariant) have the components

const Place *itsPlace and const int itsWeight, or
const Transition *itsTransition and const int itsWeight, respectively.

They are used as components of an invariant.

Lists of lists occur in the form

NL_Array = DelArray⟨N_Array⟩,
NL_Array = DelArray⟨P_Array⟩,
PI_Array = DelArray⟨P_Invariant⟩, and
TI_Array = DelArray⟨T_Invariant⟩

to build sets of node lists or invariants.

Finally, the type NPL_Array in WellStructured.h has the components

N_Array first, and N_Array second.

Both component describe two otherwise disjoint paths between the common start and end nodes first[0] = second[0], and first[first.Length ()-1] = second[second.Length ()-1].

A The equation solver gold

The program gold reads a system of equations and inequalities from an input file and writes the solution of the system to an output file. Besides several options, the filenames are read from the command line.

The first parameter (possibly extended by '.mat' for matrix) gives the input file name. If just '-' is given, the input is read from the standard input. The output and protocol file (used if the debug option is given) are specified by the next two parameters (possibly extended by '.erg' and '.prt', respectively). If these parameters are missing, the basenames of the files are taken from the input file parameter. This behaviour can be overruled by the parameter '-' which selects stdout and stderr, respectively.

The input and output file format is quite simple. Every line has the form

$$a_1 \ \ldots \ a_n \quad R \quad b$$

and describes one equation or inequality $A \cdot x \; R \; b$. R can be any of '=', '>=', or '<='. In the latter case, the inequality is replaced by $-a \cdot x \geq -b$. If one further takes an equation $a \cdot x = b$ as two inequalities $a \cdot x \geq b$ and $-a \cdot x \geq -b$, the set of equations and inequalities can be seen as one system $A \cdot x \geq b$ of inequalities.

It is also possible to replace both R and b by a question mark. Then the line is ignored in the computation, but the product $a \cdot x$ is shown in the output of the solution.

The solution is described by three sets X, Y, and Z. All vectors in them consist of integers and all vectors in Y and Z are normed such that the greatest common divisor of them is one. The meaning of the sets is best explained in the opposite order.

Z is a vector space basis of the set $\{x \mid A \cdot x = 0\}$. The elements of Y are the extremal rays of the solution cone of $\{x \mid A \cdot x \geq 0\}$ in the orthogonal space of Z. One can also describe Y as the set of solutions y for which $A \cdot y$ is of minimal non-empty support or as the smallest set such that the solution set of $A \cdot x \geq 0$ coincides with

$$\sum_{y \in Y} \mu_y \cdot y + \sum_{z \in Z} \nu_z \cdot z \qquad \text{with } \forall y \in Y\colon \mu_y \in \mathbb{Q}_+ \text{ and } \forall z \in Z\colon \nu_z \in \mathbb{Q}.$$

If the parameter '/m' (for minimal) is set, Y contains those vectors y orthogonal to Z, for which $A \cdot y$ is minimal under the non-zero non-negative solutions. This includes all vectors which would belong to Y without the parameter '/m'. In this case, all integral solutions have a representation as

$$\sum_{y \in Y} \mu_y \cdot y + \sum_{z \in Z} \nu_z \cdot z \qquad \text{with } \forall y \in Y\colon \mu_y \in \mathbb{N} \text{ and } \forall z \in Z\colon \nu_z \in \mathbb{Z}.$$

Finally, X contains all 'sufficiently small' solutions x of $A \cdot x$, in the sense that no element of Y can be subtracted from x without turning at least one component of $A \cdot x - b$ negative. Every integral solution of $A \cdot x \geq b$ can be represented as

$$x + \sum_{y \in Y} \mu_y \cdot y + \sum_{z \in Z} \nu_z \cdot z \qquad \text{with } x \in X,\ \forall y \in Y\colon \mu_y \in \mathbb{N} \text{ and } \forall z \in Z\colon \nu_z \in \mathbb{Z},$$

i.e., using exactly one element of X.

If the parameter '/r' (for rational) is set, only the corners of the base polygon of the solution set is computed. Then every rational solution is still a linear combination of the form

$$\sum_{x \in X} \lambda_x \cdot x + \sum_{y \in Y} \mu_y \cdot y + \sum_{z \in Z} \nu_z \cdot z$$

with

$$\forall x \in X\colon \lambda_x \in \mathbb{Q}_+,\ \sum_{x \in X} \lambda_x = 1,\ \forall y \in Y\colon \mu_y \in \mathbb{Q}_+,\ \text{and } \forall z \in Z\colon \nu_z \in \mathbb{Q},$$

but rational coefficients may have to be used to describe integral solutions.

Setting this parameter can improve the performance of the algorithm significantly. It is implicitly set when the system is homogenuous (as for Petri net invariants) and '`/m`' is set, because in this case X must consist of just the zero vector.

In the output, every vector x in any of these sets is displayed together with the scalar products $a \cdot x$ for all inequalities a in the system and also for all lines in the input file ending with a question mark. With '`/cols:n`' on the command line, the output width of the number representations can be set. The default is 2.

The last parameter influencing the result of the computation is '`/n`' (for non-negative). It has the same effect as including one inequalities $x_i \geq 0$ for every variable x_i in the system, but is is somewhat faster in execution. Z is always empty when this option is set.

The parameters '`/d`' (for debug) writes some protocol data about the computation into the protocol file. '`/d:2`' produces some more output.

If '`/t`' (for trace) is set, some trace information about the state of the computation are produced on the screen. The program uses the `termcap` library for this purpose and thus depends on the termcap file to be set up correctly. Again, '`/t:2`' increases the output.

Finally, '`/h`' displays a usage text on the screen and exits.

How does the program work? It first builds an equation system $A' \cdot x = 0$ by picking up all equations from A and applies the algorithm of Kannan and Bachem ([KB79]) to solve it. This algorithm has the advantage over the ordinary Gaussian elimination algorithm that the size of the numbers occurring in the computation is better controlled. Therefore it is more likely to produce small solutions.

Different to the Gaussian algorithm, it would provably run in polynomial time even if arbitrary precision arithmetic was used in the computation (which is not the case). This step can be suppressed by setting parameter '`/e`'. In that case, an equation $a \cdot x = b$ is treated similar as two inequalities $a \cdot x \geq b$ and $-a \cdot x \geq -b$.

The remaining inequalities are treated one by one. In every step considering an inequality $a \cdot x \geq b$, the solution set is intersected with $\{x \mid a \cdot x \geq b\}$. This can be done quite easily as long the scalar product $a \cdot z$ is non-zero for some element of Z.

The other case is more involved. It is handled in up to three steps.

- Firstly, the program follows an algorithm of Burger ([Bur56]) and incorporates some improvements proposed by Colom and Silva in ([CS89]) to compute the set of outer rays x of $\{x \mid A'' \cdot x \geq 0\}$, where where A'' consists of the inequalities considered so far.

- If the parameter '`/m`' is set, an exhaustive search has to be invoked to find all minimal solutions of $A'' \cdot x \geq 0$ with non-minimal support. This makes the algorithm very slow in the general case.

Fortunately, inequality systems derived from Petri net incidence matrices have very few non-zero entries normally containing small numbers. In those cases, the 'area' in which the solutions have to be sought for is very small, and the performance of the algorithm is acceptable again.

- Finally, the set X has to be updated if parameter '`/r`' is not set. Again, an exhaustive search has to be started and the remarks of the previous item apply.

References

[Aal95] W.M.P. van der Aalst. A class of Petri net for modeling and analyzing business processes. Computing Science Reports 95/26, Eindhoven University of Technology, Eindhoven, 1995.

[Aal96a] W.M.P. van der Aalst. Petri-net-based Workflow Management Software. In A. Sheth, editor, *Proceedings of the NFS Workshop on Workflow and Process Automation in Information Systems*, pages 114–118, Athens, Georgia, May 1996.

[Aal96b] W.M.P. van der Aalst. Structural Characterizations of Sound Workflow Nets. Computing Science Reports 96/23, Eindhoven University of Technology, Eindhoven, 1996.

[Aal96c] W.M.P. van der Aalst. Three Good reasons for Using a Petri-net-based Workflow Management System. In S. Navathe and T. Wakayama, editors, *Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96)*, pages 179–201, Camebridge, Massachusetts, Nov 1996.

[Aal97] W.M.P. van der Aalst. Verification of Workflow Nets. In P. Azema and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, Lecture Notes in Computer Science, page (to appear). Springer-Verlag, Berlin, 1997.

[AH97] W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Modellen, Methoden en Systemen (in Dutch).* Academic Service, Schoonhoven, 1997.

[BCH95] K. Barkaoui, J.M. Couvreur, and C. Dutheillet. On liveness in Extended Non Self-Controlling Nets. In G. De Michelis and M. Diaz, editors, *Application and Theory of Petri Nets 1995*, volume 935 of *Lecture Notes in Computer Science*, pages 25–44. Springer-Verlag, Berlin, 1995.

[Bur56] E.Burger: *Über homogene lineare Ungleichungssysteme*, Zeitung für angewandte Mathematik und Mechanik 36 (1956), 135-139.

[CS89] J.M.Colom, M.Silva: *Convex Geometry and Semiflows in P/T Nets. A Comparative Study of Algorithms for Computation of Minimal P-Semiflows*, Proceeding 10th International Conference on Application and Theory of Petri Nets (1989), 74-95.

[DE95] J. Desel and J. Esparza. *Free choice Petri nets*, volume 40 of *Cambridge tracts in theoretical computer science.* Cambridge University Press, Cambridge, 1995.

[EN93] C.A. Ellis and G.J. Nutt. Modelling and Enactment of Workflow Systems. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, Berlin, 1993.

[Eve79] S.Even: *Graph Algorithms*, Pitman, London (1979).

[ES90] J. Esparza and M. Silva. Circuits, Handles, Bridges and Nets. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 210–242. Springer-Verlag, Berlin, 1990.

[KB79] R.Kannan, A.Bachem: *Polynomial Algorithms for Computing the Smith and Hermite Normal Forms of an Integer Matrix*, Siam Journal of Computing 8 (1979), 499-507.

[MEM94] G. De Michelis, C. Ellis, and G. Memmi, editors. *Proceedings of the second Workshop on Computer-Supported Cooperative Work, Petri nets and related formalisms*, Zaragoza, Spain, June 1994.

[Mur89] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[SL96] Software-Ley. *COSA User Manual.* Software-Ley GmbH, Pullheim, 1996.

[WFM96] WFMC. Workflow Management Coalition Terminology and Glossary (WFMC-TC-1011). Technical report, Workflow Management Coalition, Brussels, 1996.

[WR96] M. Wolf and U. Reimer, editors. *Proceedings of the International Conference on Practical Aspects of Knowledge Management (PAKM'96), Workshop on Adaptive Workflow*, Basel, Switzerland, Oct 1996.

Computing Science Reports

Department of Mathematics and Computing Science
Eindhoven University of Technology

In this series appeared:

96/01	M. Voorhoeve and T. Basten	Process Algebra with Autonomous Actions, p. 12.
96/02	P. de Bra and A. Aerts	Multi-User Publishing in the Web: DreSS, A Document Repository Service Station, p. 12
96/03	W.M.P. van der Aalst	Parallel Computation of Reachable Dead States in a Free-choice Petri Net, p. 26.
96/04	S. Mauw	Example specifications in phi-SDL.
96/05	T. Basten and W.M.P. v.d. Aalst	A Process-Algebraic Approach to Life-Cycle Inheritance Inheritance = Encapsulation + Abstraction, p. 15.
96/06	W.M.P. van der Aalst and T. Basten	Life-Cycle Inheritance A Petri-Net-Based Approach, p. 18.
96/07	M. Voorhoeve	Structural Petri Net Equivalence, p. 16.
96/08	A.T.M. Aerts, P.M.E. De Bra, J.T. de Munk	OODB Support for WWW Applications: Disclosing the internal structure of Hyperdocuments, p. 14.
96/09	F. Dignum, H. Weigand, E. Verharen	A Formal Specification of Deadlines using Dynamic Deontic Logic, p. 18.
96/10	R. Bloo, H. Geuvers	Explicit Substitution: on the Edge of Strong Normalisation, p. 13.
96/11	T. Laan	AUTOMATH and Pure Type Systems, p. 30.
96/12	F. Kamareddine and T. Laan	A Correspondence between Nuprl and the Ramified Theory of Types, p. 12.
96/13	T. Borghuis	Priorean Tense Logics in Modal Pure Type Systems, p. 61
96/14	S.H.J. Bos and M.A. Reniers	The I^2 C-bus in Discrete-Time Process Algebra, p. 25.
96/15	M.A. Reniers and J.J. Vereijken	Completeness in Discrete-Time Process Algebra, p. 139.
96/17	E. Boiten and P. Hoogendijk	Nested collections and polytypism, p. 11.
96/18	P.D.V. van der Stok	Real-Time Distributed Concurrency Control Algorithms with mixed time constraints, p. 71.
96/19	M.A. Reniers	Static Semantics of Message Sequence Charts, p. 71
96/20	L. Feijs	Algebraic Specification and Simulation of Lazy Functional Programs in a concurrent Environment, p. 27.
96/21	L. Bijlsma and R. Nederpelt	Predicate calculus: concepts and misconceptions, p. 26.
96/22	M.C.A. van de Graaf and G.J. Houben	Designing Effective Workflow Management Processes, p. 22.
96/23	W.M.P. van der Aalst	Structural Characterizations of sound workflow nets, p. 22.
96/24	M. Voorhoeve and W. van der Aalst	Conservative Adaption of Workflow, p.22
96/25	M. Vaccari and R.C. Backhouse	Deriving a systolic regular language recognizer, p. 28
97/01	B. Knaack and R. Gerth	A Discretisation Method for Asynchronous Timed Systems.
97/02	J. Hooman and O. v. Roosmalen	A Programming-Language Extension for Distributed Real-Time Systems, p. 50.
97/03	J. Blanco and A. v. Deursen	Basic Conditional Process Algebra, p. 20.
97/04	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra: Absolute Time, Relative Time and Parametric Time, p. 26.
97/05	J.C.M. Baeten and J.J. Vereijken	Discrete-Time Process Algebra with Empty Process, p. 51.
97/06	M. Franssen	Tools for the Construction of Correct Programs: an Overview, p. 33.
97/07	J.C.M. Baeten and J.A. Bergstra	Bounded Stacks, Bags and Queues, p. 15.

97/08	P. Hoogendijk and R.C. Backhouse	When do datatypes commute? p. 35.
97/09	Proceedings of the Second International Workshop on Communication Modeling, Veldhoven, The Netherlands, 9-10 June, 1997.	Communication Modeling- The Language/Action Perspective, p. 147.
97/10	P.C.N. v. Gorp, E.J. Luit, D.K. Hammer and E.H.L. Aarts	Distributed real-time systems: a survey of applications and a general design model, p. 31.
97/11	A. Engels, S. Mauw and M.A. Reniers	A Hierarchy of Communication Models for Message Sequence Charts, p. 30.