Implementation of an executable graphical representation of GAPs based on Petri-nets

Sebastian Jekutsch

Project work report

Institute for Algorithms and Cognitive Systems
University Karlsruhe

Contents

1 Introduction

The contents of this work is the implementation of the bottom-up evaluation procedure of Generalized Annotated Programs (GAPs). A related procedure was presented in [2] (Chapter 3.3) and [13]. The procedure has been formulated in terms of Coloured Petri-nets [6]. Also the extension to GAP-clauses with negated body literals has been examined. The developed tool, called GAPCAD (Generalized Annotated Program Construction And Debugging), allows the interactive graphical entering of the Petri-net representation of GAPs and therefore serves as an front-end to DAEDALUS [9]. GAPCAD also permits the monitoring and step-by-step execution of GAPs. In contrast to DAEDALUS, which performs a query initiated backward chaining (SLG-resolution), the forward chaining procedure in GAPCAD computes the whole model of the GAP based on the fixpoint semantics. To compute normal GAPs, i.e. clauses with negated literals in the body, an algorithmical proposal for the computation of the well-founded model according to the alternating fixpoint characterisation [20] is presented. This will ensure answer compatibility to DAEDALUS.

The implementation uses DAEDALUS routines, a generic graph editor [5] and in-between code for representing the Petri-net and computing the fixpoint. It was taken care to define a useful interface between the GAPCAD core and the graph editor for possibly exchange with a different editor.

The outline of this report is as follows: Firstly, the generalized annotated logic, the well-founded semantics and the Coloured Petri-net formalisms are described shortly. Next, the extended Petri-net model is presented, in the first instance without negated literals and subsequently including them. Chapter 4 addresses the architecture of GAPCAD, and the final chapter discusses some further issues and an outlook. This text does not cover GAPCADs actual purpose: To serve as a front-end for developing *mediatory* knowledge bases for the integration of heterogeneous and inconsistent information sources.

2 Prerequisites

2.1 Generalized Annotated Programs

In this section the generalized annotated logic, introduced by M. Kifer and coworkers [7], is sketched. It provides an universal language for dealing with temporal, uncertain and inconsistent information or in general with parametric data with provides the algebraic structure of a lattice. For a comprehensive description of the language the reader may refer to [10, 7].

Salient features of the language are the so-called *annotations* which are constants, variables and terms over a complete lattice $\mathcal{T}$[1]. Figure 1 presents some examples for complete lattices. The following definitions are from [7]:

Definition 2.1 An *annotation* is either an element of $\mathcal{T}$ (c-annotation), an *annotation variable* (v-annotation) or a *complex annotation term* (t-annotation). Annotation terms are

[1]A complete lattice $(\mathcal{T}, \preceq)$ is a partial ordering with respect to $\preceq$, a least upper bound (lub) $\sqcup$ and a greatest lower bound (glb) $\sqcap$ for every subset of $\mathcal{T}$. A lattice is *linear* if $\preceq$ is a total ordering.

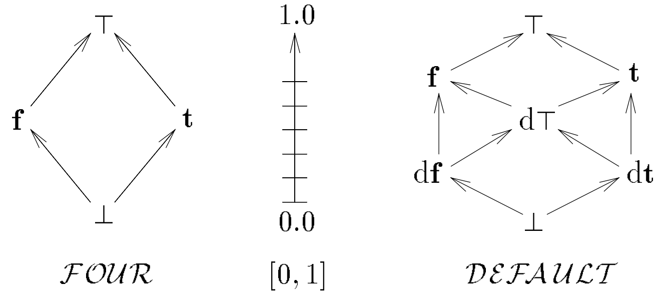


Figure 1: Some lattices used in this report

defined recursively as follows: Members of $\mathcal{T}$ and variable annotations are annotation terms. In addition, if $\mu_1, \ldots, \mu_n$ are annotation terms, then $f(\mu_1, \ldots, \mu_n)$ is a complex annotation term.

If A is a usual atomic formula of datalog (in [7] predicate calculus) and μ is an annotation, then $A : \mu$ is an *annotated atom.* An annotated atom containing no occurrence of object variables is *ground.* A is called the *object part* and μ is called the *annotation part* of $A : \mu$.

Definition 2.2 (Annotated clause) If $A : \rho$ is an annotated atom and $B_1 : \mu_1, \ldots, B_k : \mu_k$ are c- or v- annotated atoms, then

$$A : \rho \leftarrow B_1 : \mu_1 \wedge \ldots \wedge B_k : \mu_k$$

is an *annotated clause.* $A : \rho$ is called the *head* of this clause, whereas $B_1 : \mu_1, \ldots, B_k : \mu_k$ is called the *body.* All variables (object or annotation) are implicitly universally quantified. Any set of annotated clauses is called a *Generalized Annotated Program (GAP).*

Definition 2.3 (Strictly ground instance) Suppose that C is an annotated clause. A *strictly ground instance* of C is any ground instance of C that contains only c-annotations.

Let H be the Herbrand base of the program. An annotated logic interpretation I is a mapping $I : H \rightarrow \mathcal{T}$ from the base onto a lattice.

Definition 2.4 (Satisfiability) Let I be an interpretation, $\mu \in \mathcal{T}$ a c-annotation, F_1 and F_2 formulae, and A a ground atom:

1. $I \models A : \mu$ iff $I(A) \succeq \mu$.
2. $I \models \neg A : \mu$ iff $\neg(\mu) \preceq I(A)$.
3. $I \models F_1 \wedge F_2$ iff $I \models F_1$ and $I \models F_2$.
4. $I \models F_1 \vee F_2$ iff $I \models F_1$ or $I \models F_2$.
5. $I \models F_1 \leftarrow F_2$ iff $I \models F_1$ or $I \not\models F_2$.

6. $I \models F_1 \leftrightarrow F_2$ iff $I \models F_1 \leftarrow F_2$ and $I \models F_2 \leftarrow F_1$.
7. $I \models (\forall x)F$ iff $I \models \{x/t\}F$ for all ground terms t where x is an object- or annotation variable.
8. $I \models (\exists x)F$ iff $I \models \{x/t\}F$ for some ground term t where x is an object- or annotation variable.
9. If F is not a closed formula, then $I \models F$ iff $I \models (\forall)F$, where $(\forall)F$ denotes the universal closure of F.

There are two different kinds of negation in GAP, the so-called epistemic (or explicit) negation $\neg$ and the *non-monotonic* **not**. $\neg$ requires symmetry between true and false, e.g. $\neg A : \mathbf{t} = A : \mathbf{f}$ in $\mathcal{FOUR}$. The topic of non-monotonic negation will be discussed in section 2.2. For GAPs without non-monotonic negation the fixpoint operator has the following form:

Definition 2.5 (Fixpoint-operator) Let P be a generalized annotated logic program (GAP), I a GAP interpretation and $\mathcal{T}$ a complete lattice. Then a fixpoint operator $R_P(I)$ for bottom-up computation of GAPs is defined as follows: $R_P(I)(p) := \sqcup\{\rho \mid p : \rho \leftarrow p_1 : \mu_1, \ldots p_n : \mu_n$ is a strict ground instance of a clause in P and $I \models p_1 : \mu_1, \ldots p_n : \mu_n\}$.

R_P may reach the least fixpoint (lfp) if for all strict ground instances A, $lfp(R_P(A))$ is reached after a finite number of iterations. This condition, called *fixpoint reachability property* [7], holds for many GAP knowledge bases: If the clause bodies of a program contain only variable (v-) or only constant (c-) annotations, or if only finite or decreasing monotone functions[2] appear in the program. For instance, if the knowledge base consists of $Rains(Monday) : 0.5$ and $Rains(Monday) : 0.8$ the least upper bound computed by the fixpoint operator would be $Rains(Monday) : 0.8 = \sqcup\{0.5, 0.8\}$.

2.2 Well-founded semantics

For simplicity we define the well-founded semantics for classical logic in the first case, according to [19].

Definition 2.6 (Normal program) A *normal program* is a set of clauses of the form

$$A \leftarrow B_1 \wedge \ldots \wedge B_n \wedge \textbf{not } B_{n+1} \wedge \ldots \wedge \textbf{not } B_m$$

where $A, B_1, \ldots B_m$ are atoms.

Let P be a normal program and H_P its *Herbrand base* consisting of all atoms that are grounded in every possible way using all predicates, functions and constants that appear in P. For a set of literals S the expression $\neg \cdot S$ denotes the set formed by taking the complement of each literal in S. Consider as an example the following program P:

$$\begin{array}{lcl} p(a) & \leftarrow & \textbf{not } q(b) \\ q(b) & & \end{array}$$

[2]A function f is finite if $\{f(x) | x \in DOM(f)\}$ is finite and f is decreasing if for arbitrary arguments $x_1, \ldots x_n$ $f(x_1, \ldots x_n) \leq x_i$ for all $1 \leq i \leq n$.

P is a normal program with $H_P = \{p(a), p(b), q(a), q(b)\}$ and $\neg\cdot\{p(a), \neg q(a)\}=\{\neg p(a), q(b)\}$.

The well-founded model of a normal program P is a partial model[3], i.e. a set of literals which contains not necessarily all atoms of H_P. Therefore it can be seen as a three-valued model. In the above example, an interpretation $I = \{q(b), \neg q(a)\}$ states that $q(b)$ is true in I (and therefore $\neg q(b)$ is false in I), $q(a)$ is false in I and the truth values of $p(a), p(b)$ are unknown in I.

Definition 2.7 (Greatest unfounded set) Given a partial interpretation I and a normal program P, $A \subseteq H_P$ is called an *unfounded set of P with respect to I* if each atom $p \in A$ satisfies the following condition: Either there is no clause in P whose head is p, or there exists such a clause c and at least one of the following holds:

(a) some (positive or negative) subgoal of the body of c is false in I

(b) some positive subgoal of the body of c occurs in A.

The *greatest unfounded set of P with respect to I*, denoted $U_P(I)$, is the union of all unfounded sets of P w.r.t I.

In the example program P and interpretation I above, $U_P(I)$ is $\{p(b), p(a)\}$. The well-founded semantics uses $U_P(I)$ to draw negative conclusions. The transformations T_P, W_P are defined as follows:

- $T_P(I)$ is the usual fixpoint operator, i.e. $p \in T_P(I)$ iff there is some instantiated clause c of P such that c has head p and each subgoal literal in the body of c is true in interpretation I. $T_P(I)$ is called the *inner fixpoint*.
- $W_P(I) := T_P(I) \cup \neg \cdot U_P(I)$. W_P is monotonic [19].

Definition 2.8 (Well-founded model) Let $I_0 := \emptyset$, $I_{\alpha+1} := W_P(I_\alpha)$ and $I^\infty := \bigcup_\alpha I_\alpha$. I^∞ – the least fixpoint of W_P, also named *outer fixpoint* – defines the *well-founded model* of P.

The example program P has the well-founded model $\{q(b), \neg p(a), \neg q(a), \neg p(b)\}$ which is not partial. As an example for partial model consider the program which only contains the clause $p(a) \leftarrow \neg p(a)$. The well-founded model is empty, therefore the truth value of $p(a)$ is unknown.

2.2.1 Alternating fixpoint

In the following the alternating fixpoint characterisation of the well-founded model is presented shortly, according to [20]. Let $\tilde{I}$ be a set of negative literals of atoms known to be false and $P' := H_P \cup \tilde{I}$. We define $S_P(\tilde{I}) := T^{\infty}_{P'}(\emptyset)$ where $T^{\infty}_{P'}$ is the least fixpoint of $T_{P'}$, which was already defined above. $S_P(\tilde{I})$ is the set of positive facts that are derivable

[3]An interpretation or model I is seen as the set of all literals that are true in I, i.e. $\{p \in H_P \cup \neg\cdot H_P | I \models p\}$

from P and $\tilde{I}$. Let $\tilde{S}_P(\tilde{I}) := \neg \cdot (H_P \setminus S_P(\tilde{I}))$. The iteration steps $\tilde{I}_{\alpha+1} = \tilde{S}_P(\tilde{I}_\alpha)$ alternate between subsets (underestimation) of the positive portion of the partial well-founded model and supersets (overestimation) of the undefined and negative portion. The alternation converges: Let $A_P(\tilde{I}) := \tilde{S}_P(\tilde{S}_P(\tilde{I}))$, then A_P is monotone, therefore $\tilde{A} := A_P^\infty$ exists. Finally $S_P(\tilde{A}) \cup \tilde{A}$ is the well-founded model of P. The reader may refer to [20] for a deeper treatment.

Consider as an example for the alternating fixpoint computation of the well-founded model the following program taken from [20]. It describes a game where one wins if the opponent has no moves left.

Example 1

$$\begin{aligned} wins(X) &\leftarrow move(X,Y) \wedge \textbf{not } wins(Y) \\ move(a,b) & \\ move(b,a) & \\ move(b,c) & \\ move(c,d) & \end{aligned}$$

The table shows the sets S_P and $\tilde{S}_P$ at consecutive stages of the computation. They are restricted to the atoms of the *wins*-predicate, since the *move*-facts do not change during computation.

Step t	$S_P(\tilde{I}_t)$	$\tilde{S}_P(\tilde{I}_t) = \tilde{I}_{t+1}$
0	$\emptyset$	$\{\neg wins(a), \neg wins(b), \neg wins(c), \neg wins(d)\}$
1	$\{wins(c), wins(b), wins(a)\}$	$\{\neg wins(d)\}$
2	$\{wins(c)\}$	$\{\neg wins(a), \neg wins(b), \neg wins(d)\}$
3	$\{wins(c), wins(b), wins(a)\}$	Fixpoint reached

Finally the well founded model is $S_P(\tilde{I}_4) \cup \tilde{S}_P(\tilde{I}_3) = \{wins(c), \neg wins(d)\}$, restricted to the *wins* predicates.

2.2.2 Annotated logic and the well-founded model

The semantics of the non-monotonic negation in annotated logic is defined as follows:

Definition 2.9 (Satisfiability of negated atoms) Let I be an interpretation, $\mu \in \mathcal{T}$ a c-annotation and A a ground atom:

$$I \models \textbf{ not } A : \mu \text{ iff } I(A) \not\geq \mu$$

The well-founded semantics can be generalized to annotated programs. Since the semantics of the satisfiability relation $\models$ changed in annotated logic, a partial model can have a different form. Consider the following program with the lattice $[0,1]$:

$$\begin{aligned} p : 0.7 &\leftarrow \textbf{not } p : 0.5 \\ p : 0.3 & \end{aligned}$$

The partial well-founded model evaluates to $\{p : 0.3, \textbf{not } p : 0.7\}$, i.e. $p : \mu$ is true for all $\mu \preceq 0.3$, false for all $\mu \succ 0.7$ and unknown for all $0.3 \prec \mu \preceq 0.7$, where $a \succ b$ iff $a \not\preceq b$.

Example 2 reviews example 1 with annotated atoms. In the table, again only the *wins* atoms are presented. They are abbreviated in a straightforward manner, e.g. **not** $wins(a) : 0.3$ is represented as $\neg a : 0.3$.

Example 2

$$\begin{array}{lcl} wins(X) : W & \leftarrow & move(X,Y) : W \wedge \textbf{not } wins(Y) : 0.5 \\ move(a,b) : 0.3 & & \\ move(b,a) : 0.4 & & \\ move(b,c) : 0.6 & & \\ move(c,d) : 0.7 & & \end{array}$$

Step t	$S_P(\tilde{I}_t)$	$\tilde{S}_P(\tilde{I}_t) = \tilde{I}_{t+1}$
0	$\{a : 0.0,\ b : 0.0,\ c : 0.0,\ d : 0.0\}$	$\{\neg a : 0.0,\ \neg b : 0.0,\ \neg c : 0.0,\ \neg d : 0.0\}$
1	$\{a : 0.3,\ b : 0.6,\ c : 0.7,\ d : 0.0\}$	$\{\neg a : 0.3,\ \neg b : 0.6,\ \neg c : 0.7,\ \neg d : 0.0\}$
2	$\{a : 0.0,\ b : 0.4,\ c : 0.7,\ d : 0.0\}$	$\{\neg a : 0.0,\ \neg b : 0.4,\ \neg c : 0.7,\ \neg d : 0.0\}$
3	$\{a : 0.3,\ b : 0.6,\ c : 0.7,\ d : 0.0\}$	$\{\neg a : 0.3,\ \neg b : 0.4,\ \neg c : 0.7,\ \neg d : 0.0\}$
4	$\{a : 0.3,\ b : 0.6,\ c : 0.7,\ d : 0.0\}$	Fixpoint reached

The well founded model is $S_P(\tilde{I}_5) \cup \tilde{S}_P(\tilde{I}_4) = S_P(\tilde{I}_3) \cup \tilde{S}_P(\tilde{I}_3)$. This model is – different to the one in example 1 – not partial. Note that the fixpoint was reached, because step 4 results in the same sets as step 3, leading to a total model. Due to the alternating fixpoint definition, step 5 needs also be computed, because A_P evaluates two $\tilde{S}_P$-steps at a time.

2.3 Coloured Petri-nets

A coloured Petri-net is a triple $N = (P, T, A)$ consisting of disjoint sets P (*places*) and T (*transitions*) and a multiset A (*arcs*) over $(P \times T) \cup (T \times P)$ forming a bipartite graph. Each place $p \in P$ is assigned a *colourset* $C(p)$ and a multiset $M(p)$ of *tokens*, each of colour $C(p)$. Coloursets can be viewed as data types, and tokens are instances having a specific colour. Each arc $a = \langle p,t\rangle \in A$ or $\langle t,p\rangle \in A$ is attached a *label* $L(a)$ of type $C(p)$. Note that tokens as well as labels may contain variables of suitable type. A *marking* is the distribution of tokens over all places of the net. Each transition $t \in T$ is assigned a *Boolean guard* $G(t)$ expressing constraints on the variables binded to t. For an extended and more formal definition of coloured Petri-nets, the reader may refer to [6].

Let $IN(t) := \{\langle p,t\rangle \in A \mid p \in P\}$, $OUT(t) := \{\langle t,p\rangle \in A \mid p \in P\}$, $\bullet t := \{p \mid \langle p,t\rangle \in A\}$ and $t\bullet := \{p \mid \langle t,p\rangle \in A\}$ denote the vicinity of $t \in T$. A transition t is called *enabled* iff the following conditions hold:

- For each incoming arc $a_i = \langle p,t\rangle \in IN(t)$ there is at least one variable substitution σ_i, such that a token $s \in M(p)$ exists with $\sigma_i(s) = \sigma_i(L(a_i))$. This particular token s must not serve again as a resource for another substitution σ_j for $j \neq i$. Recall that $M(p)$ is a multiset, therefore more than one token of this kind may be present.

- All substitutions σ_i $(1 \le i \le |IN(t)|)$ are compatible. σ_i and σ_j are compatible if their concatenation $\sigma_i\sigma_j$ is defined. In other words, there is no assignment of two different values to the same variable.

- $G(t)$ evaluates to true under $\sigma = \sigma_1\sigma_2\cdots\sigma_m$. In this case, σ is called an *enabling substitution.*

There could be more than one enabling substitution under the same marking. A transition could *fire*, if it is enabled under a substitution σ. If a transition t fires, the tokens M_i of the places are updated to M_{i+1} as follows:

$$M_{i+1}(p) := \begin{cases} M_i(p) \setminus \sigma(L(\langle p,t\rangle)) & \text{if } p \in \bullet t \setminus t\bullet \\ M_i(p) \cup \sigma(L(\langle t,p\rangle)) & \text{if } p \in t\bullet \setminus \bullet t \\ \{M_i(p) \setminus \sigma(L(\langle p,t\rangle))\} \cup \sigma(L(\langle t,p\rangle)) & \text{if } p \in \bullet t \cap t\bullet \\ M_i(p) & \text{otherwise} \end{cases}$$

Given a marking M_0, a sequence $t_1, \ldots t_n$ is called a *firing sequence*, if for each i $(1 \le i \le n)$, it holds that $t_i \in T$ is enabled under the marking M_{i-1} and t_i's firing results in the marking M_i. The firing sequence changes the marking M_0 into M_n.

3 An extended Petri-net Model

3.1 Negation-free GAPs

A GAP knowledge base is transformed into an extended Petri-net $N = (P, T, A)$ according to the subsequent rules (suppose the clauses are enumerated from 1 to n):

- Each predicate p is a place $p \in P$ in the net.

- Each clause c $(1 \le c \le n)$ is a transition $c \in T$ in the net.

- Let O be the type of the object part and $\mathcal{T}$ the annotational lattice of predicate p. Then $C(p) := O \times \mathcal{T}$.

- For every clause c of the form

$$p_0(o_0) : \mu_0 \leftarrow p_1(o_1) : \mu_1 \wedge \ldots \wedge p_m(o_m) : \mu_m$$

  and $1 \le i \le m$, the net contains the arcs $a_i := \langle p_i, c\rangle$ with the labels $L(a_i) := (o_i, \bar{\mu_i})$, where $\bar{\mu_i}$ is a new variable annotation. If μ_i is a c-annotation, then $\mu_i \preceq \bar{\mu_i}$ is added as a conjunctional condition to the guard of transition c. In addition, the net contains the arc $a_0 := \langle c, p_0\rangle$ with the label $L(a_0) := (o_0, \bar{\mu_0})$, where

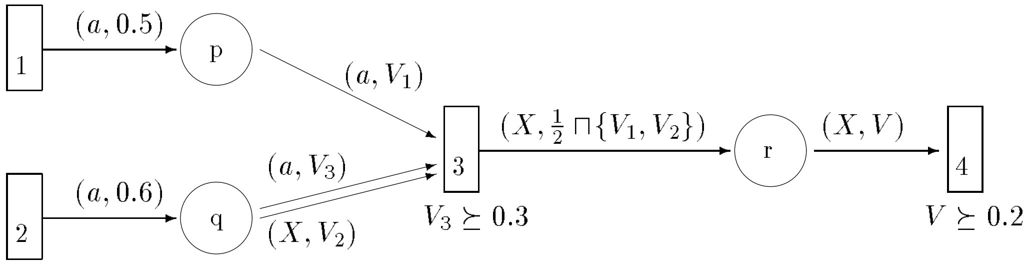
$$\bar{\mu_0} := \begin{cases} \mu_0 & \text{if } \mu_0 \text{ is c-annotation} \\ \sqcap\{\bar{\mu_i} | \mu_i \text{ is the same variable as } \mu_0\} & \text{if } \mu_0 \text{ is v-annotation} \\ f(\bar{\rho_1}, \ldots \bar{\rho_n}) & \text{if } \mu_0 = f(\rho_1, \ldots \rho_n) \end{cases}$$

  The $\bar{\rho_1}, \ldots \bar{\rho_n}$ are defined recursively in the same way. Note that $\sqcap\{a\} = a$ for every $a \in \mathcal{T}$ and $\sqcap\{\} := \sqcap\mathcal{T}$.

- The initial marking is $\forall p \in P : M_0(p) = \{(X, \sqcap\mathcal{T})\}$, where X is a new variable for all p and $(X, \sqcap\mathcal{T}) \in C(p)$.

Queries can be added to the net, as they are headless clauses. The following abstract example illustrates the transformation in its details. Places are drawn as circles and transitions as rectangles. Typing information is omitted and $C(p) = C(q) = C(r) = \{a\} \times [0,1]$. All uppercase letters are variables.

Example 3 *(1)* $p(a) : 0.5 \leftarrow$
(2) $q(a) : 0.6 \leftarrow$
(3) $r(X) : \frac{1}{2}V \leftarrow p(a){:}V \wedge q(X){:}V \wedge q(a){:}0.3$
(4) $\leftarrow r(X) : 0.2$



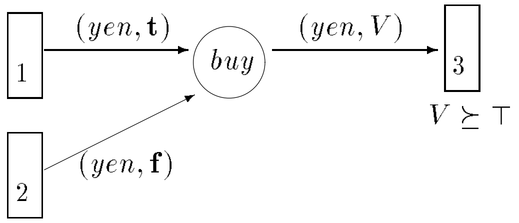
In the following, a substitution is written as a set of bindings of the form X/t, where X is a variable and t is a term of appropriate colour. In example 3, the answering of the query $r(X) : 0.2$ can be modelled by a firing sequence $1, 2, 3, 4$: Transitions 1 and 2 are always enabled since their guards are true and no variable binding is necessary. Their firing places the token $(a, 0.5)$ in p and $(a, 0.6)$ in q. Consider now transition 3: A possible substitution is $\sigma = \{V_1/0.5, V_2/0.6, V_3/0.6, X/a\}$. Due to the fact that the guard $V_3 \succeq 0.3$ evaluates to true under σ, transition 3 is enabled. Its firing (see below for problems here) adds the token $\sigma((X, 0.5 \cdot \sqcap\{V_1, V_2\})) = (a, 0.25)$ to place r. Finally the query transition 4 is enabled with $\sigma = \{X/a, V/0.25\}$, which is also the substitution for the successful query.

We need to extend the model in the following three ways, in order to capture the fixpoint semantics:

1. In the example above, only one token was in place q after transition 2 fired, but transition 3 needed this token *two* times to be enabled, one for every arc $\langle \mathrm{q}, 3\rangle$. Unlike the definition in section 2.3, tokens will *not* be removed in our model if a transition fires. This reflects the fact that the tokens represent knowledge, rather than resources that cannot be shared. In other words, our Petri-net model caches all facts necessary for answering a query, which could lead to a large number of tokens to be kept within the net. Such an extension avoids conflicts between transitions which need the same token to be enabled, as encountered in the example.

2. The model presented so far only works with linear annotation lattices. Consider the following example using the non-linear lattice $\mathcal{FOUR}$.

Example 4 *(1)* $buy(yen) : \mathbf{t} \leftarrow$
(2) $buy(yen) : \mathbf{f} \leftarrow$
(3) $\leftarrow buy(yen) : \top$



After the firings of 1 and 2, *buy* contains the tokens $(yen, \mathbf{t})$ and $(yen, \mathbf{f})$. There are two possible substitutions for V: $\{V/\mathbf{t}\}$ and $\{V/\mathbf{f}\}$. None of them satisfies the guard $V \succeq \top$, hence transition 3 is not enabled. This is a contradiction to the fixpoint semantics of GAPs, because $\sqcup\{\mathbf{t}, \mathbf{f}\} = \top$. In the example, a token $(yen, \top)$ should be in $M(p)$ although none of the incoming transitions 1 and 2 delivered it. We call such derived tokens *reductants* [7][4].

Definition 3.1 (Reductants) Given a set $M = \{(o_1, \mu_1), \ldots (o_n, \mu_n)\}$ of tokens and a unification σ with $\sigma(o_1) = \ldots = \sigma(o_n)$, the token $(\sigma(o_1), \sqcup\{\mu_1, \ldots \mu_n\})$ is called a *reductant.* The function *reductants(M)* computes the set of reductants derived from all subsets of M for which σ is defined.

For example $((a, b), \top)$ is a reductant of the set $\{$((X,b),$\mathbf{t}$), ((a,Y),$\mathbf{f}$)$\}$. It is important that every annotation in M is a c-annotation to ensure that the least upper bound $\sqcup$ is defined. For markings $M(p)$ this is always the case according to the next theorem. A proof has appeared in [2].

Theorem 1 (Possible tokens of a place) Let P be a GAP and N its transformation. At all places $p \in P$ of $N = (T, P, A)$, there are only tokens $(o, \mu) \in M(p)$ with $\mu \in \mathcal{T}$, if P is finite.

3. It is also possible to delete tokens from a place. For example, every time $(a, 0.5) \in M(p)$ serves as a token for an enabling substitution of transition t (with $\langle p, t\rangle \in A$), $(a, 0.6)$ will as well; but not vice versa. We say that $(a, 0.6)$ *subsumes* $(a, 0.5)$, because $0.6 \succeq 0.5$ in the lattice $[0, 1]$. $(a, 0.5)$ might be deleted from $M(p)$ without changing the behaviour of the extended Petri-net.

Definition 3.2 (Subsumption) Given two tokens $(o_1, \mu_1), (o_2, \mu_2) \in M$, the first *subsumes* the second if $\mu_1 \succeq \mu_2$ and there exists a substitution σ such that $o_2 = \sigma(o_1)$. The function *subsumption(M)* computes all tokens in M which are subsumed by at least one other token in M.

For example $(a, \mathbf{t})$ and $(a, \mathbf{f})$ are both subsumed by their reductant $(a, \top)$, whereas $subsumption(\{((X, b), \mathbf{t}), ((a, Y), \mathbf{f}), ((a, b), \top)\})$ is empty.

[4]Different from the definition provided here, in [7] derived *rules* are named reductants. Note that tokens are representations for annotated atoms due to the presented transformation.

To summarize the three extensions presented above, we redefine the update M_{i+1} of the marking M_i due to the firing of transition $t \in T$:

$$\begin{array}{lll} (1) & M^{up}_{i+1}(p) & := \begin{cases} M_i(p) \cup \sigma(L(\langle t, p\rangle)) & \text{if } p \in t\bullet \\ M_i(p) & \text{otherwise} \end{cases} \\ (2) & M^{red}_{i+1}(p) & := M^{up}_{i+1}(p) \cup \mathit{reductants}(M^{up}_{i+1}(p)) \\ (3) & M_{i+1}(p) & := M^{red}_{i+1}(p) \setminus \mathit{subsumption}(M^{red}_{i+1}(p)) \end{array}$$

With this extension, example 4 works as expected: Transitions 1 and 2 place the tokens $(yen, \mathbf{t})$ and $(yen, \mathbf{f})$ in p respectively. $M^{red}(p)$ evaluates to $\{(yen, \mathbf{t}), (yen, \mathbf{f}), (yen, \top)\}$ and $M(p)$ to $\{(yen, \top)\}$, which enables transition 3, since $\top \succeq \top$.

It is worth noting that our model captures the operational semantics of a GAP, which means that if there is a GAP for which the least fixedpoint reachability property does not hold (e.g. from $\{p : 0, p : \frac{1+x}{2} \leftarrow p : x, q : 1 \leftarrow p : 1\}$ it is never possible to answer the query $q : 1$) the corresponding Petri-net cannot answer this query as well and runs forever.

The following theorems have been proven in [2] and capture the soundness and completeness of the proposed extended Petri-net model with respect to the semantics of GAPs:

Theorem 2 (Soundness) Let P be a GAP with clauses $c_1, \ldots c_n$, c_n a query and N the extended Petri-net defined on P. If there is a successful firing sequence in N then $c_1, \ldots c_{n-1} \models c_n$.

Theorem 3 (Completeness) Let P be a GAP with clauses $c_1, \ldots c_n$, c_n a query and N the extended Petri-net defined on P. If $c_1, \ldots c_{n-1} \models c_n$, then there is a successful firing sequence in N.

3.1.1 Algorithms for the extended Petri-net model

Before presenting algorithms for the testing for fireability of a transition and updating of the net marking, some more definitions are required.

Definition 3.3 (Unifier $mgu_{\mathbf{a}}()$ of tokens) Tokens $s = (o, \mu)$ as well as arc labels consist of two parts, its first being the object part $s^{obj} = o$ and its second being the annotation part $s^{ann} = \mu$. Let $mgu(o_1, o_2)$ denote the usual most general unifier of o_1 and o_2. Given two tokens/labels s_1, s_2, the *most general annotational unifier*, denoted $mgu_{\mathbf{a}}(s_1, s_2)$, is defined as follows:

$$mgu_{\mathbf{a}}(s_1, s_2) := \begin{cases} mgu(s_1^{obj}, s_2^{obj}) \cup \{s_2^{ann}/s_1^{ann}\} & \text{if } s_2^{ann} \text{ is v-annotation} \\ mgu(s_1^{obj}, s_2^{obj}) & \text{if } s_1^{ann} \text{ and } s_2^{ann} \text{ are c-annotations} \\ & \quad \text{and } s_1^{ann} \succeq s_2^{ann} \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

If $mgu(s_1^{obj}, s_2^{obj})$ is not defined, $mgu_{\mathbf{a}}(s_1, s_2)$ is not defined either. Note that $mgu_{\mathbf{a}}()$ is not symmetric. If $mgu_{\mathbf{a}}(s_1, s_2)$ is defined, then s_1 and s_2 are said to be *unifiable*.

For example, $mgu_{\mathbf{a}}(((X,a),0.5),((b,Y),0.4)) = \{X/b, Y/a\}$ and $mgu_{\mathbf{a}}((a,\mathbf{t}),(a,\mathbf{f}))$ is not defined.

Definition 3.4 (Compatibility of annotation substitutions) Two substitutions $\{V/a\}$ and $\{V/b\}$, which assign different c-annotations a and b to the same annotation variable V, are *compatible* if a and b are comparable due to the ordering of the underling lattice. In this case their *concatenation* $\{V/a\}\{V/b\}$ is defined as $\{V/\sqcap\{a,b\}\}$. This definition is easily extended to cases with more than two substitutions.

Definition 3.5 (Concatenation of $mgu_{\mathbf{a}}$s) The substitutions in $mgu_{\mathbf{a}}$s may be divided in object variable substitutions and annotation variable substitutions. The *concatenation* $\sigma_1\sigma_2\cdots\sigma_n$ of n $mgu_{\mathbf{a}}$s $\sigma_1,\ldots\sigma_n$ is defined as the usual concatenation of the object variable substitutions unioned with the above defined concatenation of the annotation variable substitutions.

The following algorithms do *not* use the notion of transition guards. Instead they are specialized for the bottom-up evaluation of annotated programs encoded in Petri-nets. The guards are implicitly checked in the $mgu_{\mathbf{a}}$-routine defined above.

Testing for fireability of a transition

Input: Extended Petri-net $N = (P, T, A)$;
Transition $c \in T$
Output: Maximal set of $mgu_{\mathbf{a}}$s each enabling c. c is not enabled if the set is empty.

```
$\Theta := \{\}$;          *($\Theta$ is a set of sets of possible substitutions for $c$)*
**for all** arcs $a \in IN(c)$ **do**
        *(Let $a$ be $(p,c) \in A$)*
        $\varphi_a := \{\}$;   *($\varphi_a$ is the set of all possible substitutions according to $a$)*
        **for all** tokens $s \in M(p)$ **do**
                **if** $unifiable(s, L(a))$ **then** $\varphi_a := \varphi_a \cup mgu_{\mathbf{a}}(s, L(a))$;
        **if** $\varphi_a = \{\}$ **then return** $\{\}$;
        $\Theta := \Theta \cup \varphi_a$;
*(Let $\Theta$ be $\{\varphi_1, \ldots \varphi_{|\Theta|}\}$)*
$\Psi := \{\}$;          *($\Psi$ is a set of all enabling substitutions for $c$)*
**for all** permutations $(\sigma_1, \ldots \sigma_{|\Theta|})$ with $\sigma_i \in \varphi_i \in \Theta$ $(1 \le i \le |\Theta|)$ **do**
        **if** $\sigma_1, \ldots \sigma_{|\Theta|}$ are compatible in pairs **then** $\Psi := \Psi \cup \sigma_1\sigma_2\cdots\sigma_{|\Theta|}$;
**return** $\Psi$;
```

Firing of a transition

Input: Extended Petri-net $N = (P, T, A)$;
Transition $c \in T$;
Set Ψ of c-enabling substitutions
Output: N with updated marking using every $\sigma \in \Psi$

```
**for all** arcs $a \in OUT(c)$ **do**
    *(Let $a$ be $(c, p) \in A$)*
    **for all** $\sigma \in \Psi$ **do**
        $M(p) := M(p) \cup \{\sigma(L(a))\}$;
    $M(p) := M(p) \cup reductants(M(p))$;
    $M(p) := M(p) \setminus subsumption(M(p))$;
```

See [9] for algorithms implementing *reductants*() and *subsumption*(). The algorithms are based on the rather descriptive than procedural algorithms in [2], which need two minor corrections: Firstly, in [2] the initial marking is empty. This leads to incompleteness, because even without any fact, the bottom element $P : \sqcap T$ of any predicate p is derivable. Secondly, the testing for fireability on page 19 needs to be modified in step 2 in the following way:

2. Compute for all i $(1 \leq i \leq n)$ sets $M_i^j = \{(t_1^{P_i}, \mu_1^{P_i}, \ldots (t_{n_i^j}^{P_i}, \mu_{n_i^j}^{P_i})\}$ of tokens of the place P_i $(e_i = \langle P_i, k\rangle)$ and a substitution Φ_i^j for every set M_i^j such that the following conditions hold:
   (a) $\Phi_i^j = unifier(t_i, t_1^{P_i}, \ldots t_{n_i}^{P_i})$
   (b) $c_i \leq \sqcup(\mu_1^{P_i}, \ldots \mu_{n_i}^{P_i})$
   (c) $\Phi_i = \Phi_i^1 \cdots \Phi_i^j$ if $\Phi_i^1, \ldots \Phi_i^j$ are compatible in pairs.
   (d) $\Phi = \Phi_1 \cdots \Phi_i$ if $\Phi_1, \ldots \Phi_i$ are compatible in pairs.
   (e) There is no $\Phi' > \Phi$ which satisfies (a)-(d).

The main procedure checks in each step for an arbitrary transition t whether any of its input places $p \in \bullet t$ contains new tokens relative to the last step. If not, it checks another transition and stops, if no transition satisfies this condition, because this means, that no new useful token was produced in the last step, therefore the fixpoint is reached. If on the other hand a transition is found, it is fired if it is enabled. Then the next step is taken. There is an indeterminism in choosing an arbitrary transition in the beginning of each new step. This choice should be fair, i.e. *every* transition is checked a finite count of steps after it has been checked last. See also section 5.1 for a discussion of this point.

Main procedure

Input: Extended Petri-net $N = (P, T, A)$ with initial marking
Output: N with marking which represents the fixpoint

```
**iterate** through all $t \in T$
    **if** $\exists p \in \bullet t$ with new tokens since last firing of a transition **then**
        **if** $t$ is enabled **then**
            fire $t$;
            restart iteration;
```

3.2 Normal GAPs

This section describes a method how to handle non-monotonic modes of negation based upon the well-founded semantics in the extended Petri-net formalism, using a direct implementation of the alternating fixpoint computation. The reader may also refer to [8] for a similar presentation not based on Petri-nets.

The transformation of a *normal* GAP to an extended Petri-net $N = (P, T, A)$ is performed as follows: Every negated atom of the form **not** p is treated as a new, not negated atom '*not_p*', i.e. for every predicate a dual negative predicate is added to the vocabulary. This transforms the normal GAP into a negation-free GAP. The set of places P is therefore divided into two sets: P^+, containing the positive literals, and P^-, containing the previously negative literals, such that $P = P^+ \cup P^-$. The transformation process is similar to the one described in section 3.1, but with the following differences:

- For every clause c of the form

$$p_0(o_0) : \mu_0 \leftarrow p_1(o_1) : \mu_1 \wedge \ldots \wedge p_m(o_m) : \mu_m$$

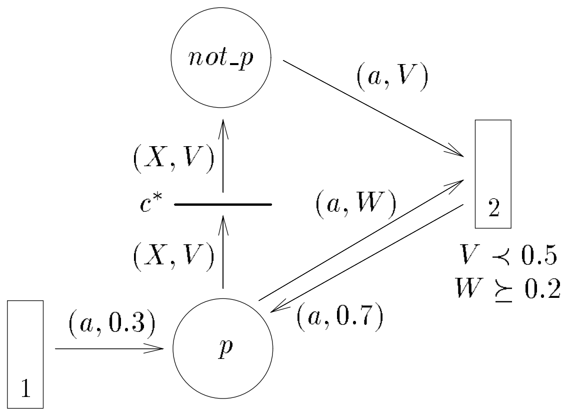
  and $1 \leq i \leq m$, the net contains the arcs $a_i := \langle p_i, c \rangle$ with the labels $L(a_i) := (o_i, \bar{\mu}_i)$ where $\bar{\mu}_i$ is a new variable annotation. In case that μ_i is a c-annotation: If $p_i \in P^+$, then $\mu_i \preceq \bar{\mu}_i$ is added as a conjunctional condition to the guard of transition c; **otherwise if** $p_i \in P^-$, **then** $\mu_i \succ \bar{\mu}_i$ **is added to the guard.** In addition, the net contains the arc $a_0 := \langle c, p_0 \rangle$ with the label $L(a_0) := (o_0, \bar{\mu}_0)$ where $\bar{\mu}_0$ is defined as in section 3.1.

- The initial marking is $\forall p \in P^+ : M_0(p) = \{(X, \sqcap\mathcal{T})\}$, **and** $\forall p \in P^- : M_0(p) = \{(X, \sqcup\mathcal{T})\}$, where X is a new variable for each p.

Additionally, new transitions T^* will be introduced, as shown in example 5:

- Given two dual places $p \in P^+$ and *not_p* $\in P^-$, a transition $c^* \in T^*$ is added to T with an empty guard. The arcs $\langle p, c^* \rangle$ and $\langle c^*, not\_p \rangle$, both with labels (X, V) with new variables X and V, are added to A. If such a transition fires, all tokens from p are moved to *not_p*.

Example 5 *(1)* $p(a) : 0.3 \leftarrow$
(2) $p(a) : 0.7 \leftarrow not\_p(a) : 0.5 \wedge p(a) : 0.2$



The following algorithm schema describes, how to compute the well-founded model:

1. Compute the fixpoint as described in section 3.1 without firing any transition from T^*. This realizes S_P.
2. Transfer all tokens along the transitions in T^*, i.e.:
   - Delete all tokens out of places in P^-.
   - Fire every transition in T^*.
   - Delete all tokens out of places in P^+ and restore the initial marking in these places.

   This performs $\tilde{S}_P$.
3. Redo these two steps. This results in $\tilde{S}_P(\tilde{S}_P)$.
4. If the outer fixpoint is not reached, go to step 1. To test this, the marking needs to be saved to compare it to the new marking after step 1.
5. The fixpoint is reached. Take step 1 one more time. The resulting marking represents the well-founded model.

The following table demonstrates the algorithm referring to example 5:

Step	Marking	
	$p(X) : 0.0,\ not\_p(X) : 1.0$	Initialisation
1	$p(a) : 0.3,\ not\_p(X) : 1.0$	(1) fires
2	$p(X) : 0.0,\ not\_p(a) : 0.3$	(c^*) fires
1	$p(a) : 0.7,\ not\_p(a) : 0.3$	(1)+(2) fire
2	$p(X) : 0.0,\ not\_p(a) : 0.7$	No fixpoint
1	$p(a) : 0.3,\ not\_p(X) : 1.0$	The same four steps repeated
2	$p(X) : 0.0,\ not\_p(a) : 0.3$	
1	$p(a) : 0.7,\ not\_p(a) : 0.3$	
2	$p(X) : 0.0,\ not\_p(a) : 0.7$	Fixpoint reached
1	$p(a) : 0.3,\ not\_p(a) : 0.7$	Well-founded model

This procedure has some problems: Tokens need to be deleted explicitly and two different firing semantics need to be observed. Also transitions are locked and unlocked periodically. After all, this results in something very different from Petri-nets. To check whether the fixpoint was reached, many tokens need to be remembered and compared. To implement this, it will be best to divide every place from P^+ in two parts, one storing $\tilde{S}$, the other $\tilde{S}(\tilde{S})$. The solution in [8] bares the same disadvantages.

An alternative solution is described in [16]. This is more closely to Petri-nets, but adds inhibitor arcs and requires the explicit computation of the greatest deadlock of the net to obtain the unfounded set of a program: A non-empty subset $S \subseteq P$ of places in a Petri-net is called a *deadlock*, if every transition having an output place in S has an input place in S. Note that in this approach enabling tokens *will* be removed if a transition fires, according to the definition of the update of markings in ordinary Petri-nets. The greatest disadvantage of the approach in [16] is the translation schema: Every place represents a *ground* atom rather than a predicate, which makes it of little use for graphical applications addressed in this report.

To summarize, there is no known elegant method for Petri-net computation of the well-founded model of normal programs. In the next section, a tool implementing the routines in section 3.1 is presented. It does *not* allow normal programs and therefore does not implement the well-founded semantics, due to the problems encountered above.

4 GAPCAD - Architecture

GAPCAD is the implementation of the procedures presented in section 3.1. This chapter briefly reviews the concepts behind GAPCAD.

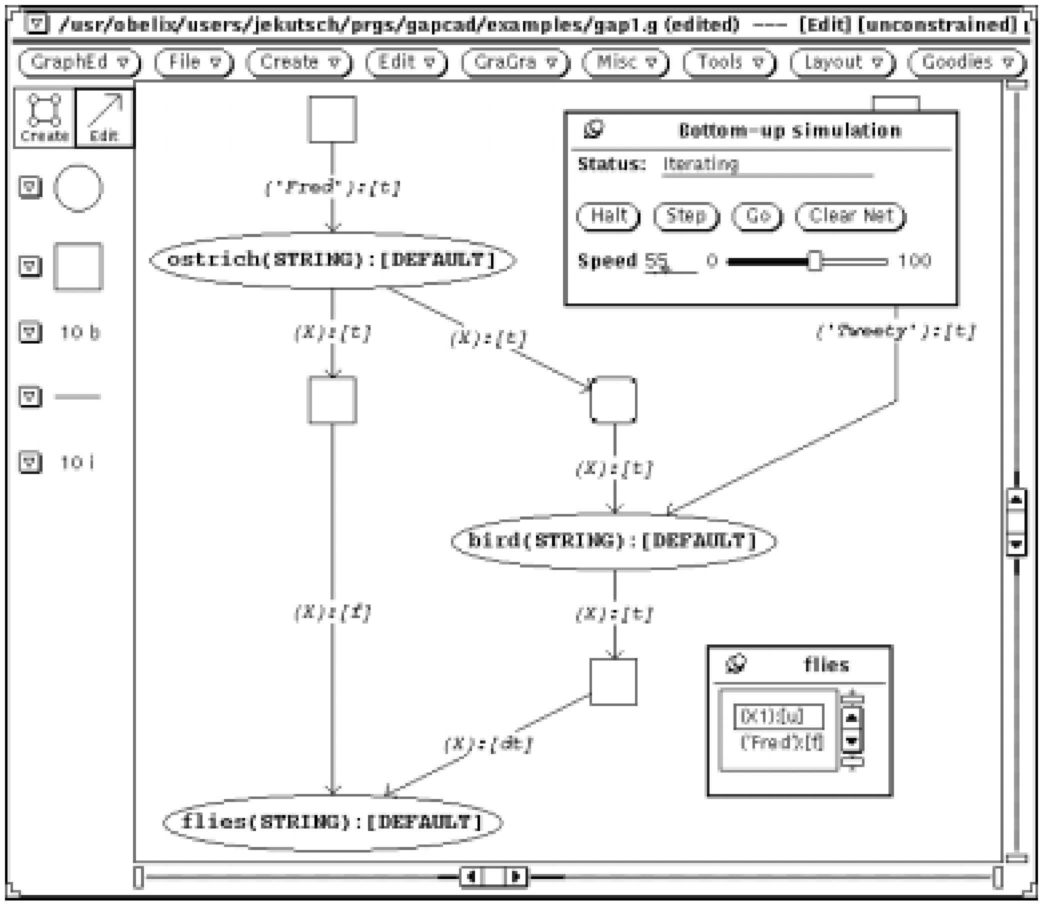


Figure 2: Screendump of the GAPCAD user interface; with a well-known problem. The current tokens in place `flies` are shown at the right bottom. At the top, a control panel presents the features described in the text

GAPCAD offers the following features:

- One can draw a Petri-net and save it as a kind of painting, or load a GAP-program in DAEDALUS syntax. In the latter case the net is automatically being drawn in the window as a Petri-net. Guards need not be typed. They are automatically derived from the arc labels.

- After finishing the drawing, the graph needs to be compiled into the internal extended Petri-net structure. Syntax errors are located.
- After compiling, one can
  - start the bottom-up procedure described in chapter 3.
  - start DAEDALUS, assuming a query (transitions without outgoing arcs) was entered.
  - save the net as a GAP in DAEDALUS syntax.
- During the bottom-up procedure
  - every firing transition highlights.
  - the tokens in any place are shown, if requested.
  - a transition can manually be forced to fire.
- The features of the graph editor are preserved.

See the GAPCAD manual included in the distribution for detailed descriptions.

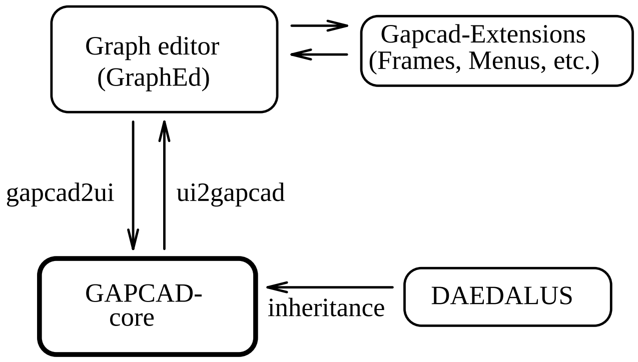


Figure 3: The GAPCAD architecture

GAPCAD is a graphical user interface to DAEDALUS. Therefore the system can be divided in three parts as illustrated in figure 3:

1. DAEDALUS [9] provides generic object data classes and lattice classes, and concepts like predicates, literals, substitutions, etc. GAPCAD uses heavily DAEDALUS-code for those basic GAP-functions like unifying, computing the least upper bound and so on.
2. The graph editor provides a front-end for entering Petri-nets. GraphEd [5] was chosen, because

- it is generic in the sense that it provides application interfaces for adding domain specific functionality.
- it is easy to use.
- it is public domain.

Unfortunately some code had to be added in the core of GraphEd to provide an even easier entering of Petri-nets, for example using the left and right mouse buttons to create places and transitions or suppressing attempts to connect two places. It was also important to ensure consistency between the GraphEd- and the GAPCAD data structures, for example token windows need to be deleted as soon as the corresponding place is deleted.

3. The GAPCAD-core itself has its own data structure. While running the bottom-up procedure it needs to be consistent with the GraphEd data structure (for example to highlight a transition) as well as the DAEDALUS data structure (for example to use the routines for unifying). Figure 4 shows the relevant classes.

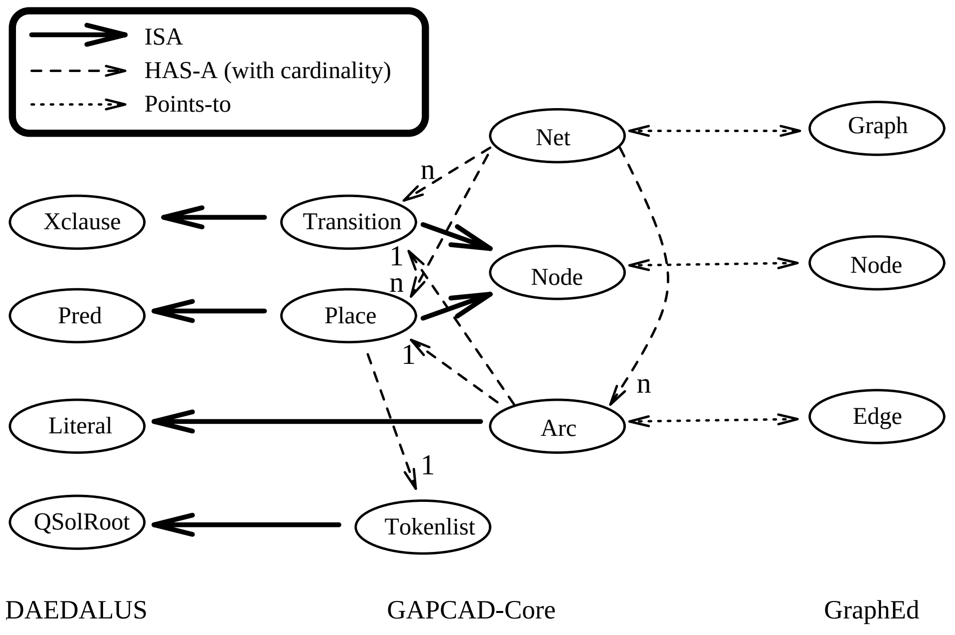


Figure 4: The GAPCAD class structure

One goal was to separate the graph editor from the GAPCAD-core as far as possible to make it easily possible to use another editor. Two interfaces were defined:

`gapcad2ui`: This contains procedures which GAPCAD provides for the editor such as creating and deleting nets/places/transitions/arcs, firing of transitions, starting Daedalus/bottom-up evaluation, loading/saving/printing the GAP, etc.

`ui2gapcad`: This specifies services which the editor needs to provide, such as displaying new nodes and edges, highlighting of a node, setting labels or refreshing token windows.

Some extensions of GAPCAD would be useful:

- Currently no parallelizing is supported. Every enabled transition is immediately fired. It would be interesting to compute the conflict set of transitions which are enabled; see also chapter 5.1. It would be easy to implement this, because there are two different procedures for checking for fireability and firing of a transition.
- For a more efficient computation, it would be useful to switch of the graphical representation completely.
- It could be interesting to examine the firing sequence as a list.
- In [23] and [12] a Petri-net based validation check of rule-based programs is described. Integration in GAPCAD should be easy because of the simple class structure.
- Unfortunately GraphEd provides no graph overview facility. For large GAPs the graph becomes too complicated. Also the automated graph drawing of a loaded GAP is far from optimal.
- For more easy entering of graphs, it would be useful to implement hierarchical coloured Petri nets [6].

5 Further issues

In this chapter, two further subjects are addressed: 1. control flow specification and 2. GAPCAD as a knowledge acquisition tool.

5.1 Control flow specification

From a software engineering point of view, the extended Petri-net model specifies the data flow. The places are data containers and the transitions represent the operations on the data, especially if the guards contain more complex functions. On the other side *no* control flow specification is given. Each enabled transition may (but need not) fire, which causes indeterminism. This is sound according to the fixpoint-semantics [2]. A specification of the control flow, i.e. the determination which of the enabled transition fire in any state of the net, is not necessary. It could even make the extended Petri-net model incomplete, if it never allows a particular transition to fire, which contribution to the fixpoint set is not empty. This shows that a control flow specification needs to satisfy some conditions. For example it needs to be *fair*, i.e. every enabled transition fires sometime until the fixpoint is reached.

However, explicitly specifying the control flow has two advantages:

1. Efficiency: Even if a transition is enabled, its firing does not necessarily enlarge the fixpoint set. If a query has been stated, the point becomes obvious. Consider as an example the following program:

$$\begin{array}{rcll} p(X) & \leftarrow & q(X) & \qquad (*) \\ q(a) & & & \\ q(b) & & & \end{array}$$

$$q(c)$$
$$\vdots$$
$$Query: \quad p(a)$$

The query is answered in two steps, but a pure bottom-up procedure computes the whole model. A possible way to resolve this overhead is the magic set approach [1], where the clause $(*)$ would be rewritten to $p(a) \leftarrow q(a)$, depending on the known query.

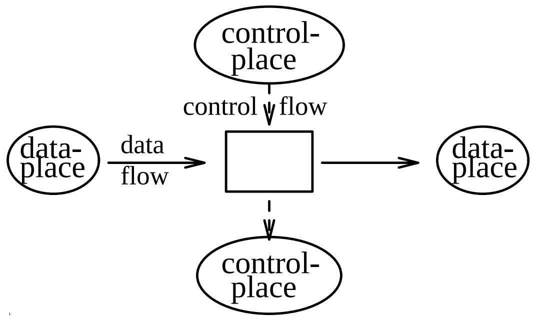
2. Side effects: Consider the case where the firing of a transition causes a side effect output to the screen. In the most cases the user is only interested in the final solution, not in several in-between results, so this transition should fire as late as possible. This can only be achieved if the control flow is explicitly specified. Of course, this argument is not a theoretical but a practical one.

There are at least three possibilities to describe the control flow:

Firm strategy Production systems, which also work in a bottom-up manner, usually resolve the conflict that arises if more then one rule at the time is ready to fire, via some heuristics like "Take the most recent enabled rule" or "Take the rule with the largest number of premises" or so. In GAPCAD a rather simple conflict resolution is implemented: It iterates through the set of transitions. If the current transition t is enabled *and* some new token is in any of the places in $\bullet t$, it fires and the iteration restarts. This algorithm terminates because the fixpoint is reachable and its occurrence causes *no* new token being in any place.

There is one major disadvantage in completely specifying the control flow: Two transitions being enabled at the same time express the possibility to fire them in parallel. There is not always the need to completely sequentialize the order of firing. In the case of Petri-nets, a control specification should be better viewed as a *restriction* of freedom, rather than a total sequentialisation. The next two techniques take this into account.

Petri-nets We used the Petri-net model to define the data flow, although Petri-nets usually specify the control flow. It is possible to unify both applications: Orthogonally to the



data flow we embed the transitions in a second Petri-net where the places contain control tokens. A transition may be enabled only if its input control-places contain tokens. In [3] it is shown that as soon as the control flow gets more complex, Petri-nets tend to be difficult to survey, so this kind of control flow specification appears to be not very natural.

Priorities In [3] it is suggested to express the control flow through dynamically given priorities between transitions or temporally locks of transitions. If two transitions are enabled, the one with the higher priority fires (if it is not locked), while the other has to wait. If the two have the same priority they do both fire in parallel. These priorities may change on certain events like a counter reaching zero, an external condition becoming true, time constraints, etc. The reader may refer to [3] for more ideas. This seems to me the most promising approach for control specification, although dynamically changing priorities may lead to confusion about what state the net is currently in.

5.2 GAPCAD as a knowledge acquisition tool

KADS (Knowledge Acquisition and Design Structuring) [22] is a methodology for developing knowledge-based systems (KBS). Its emphasis lies on defining a language for semi-formal specification of KBS. Neither any knowledge elicitation technique nor implementational details are covered. KADS offers some abstract templates (called *models*) which need to be filled during the specification phase. A central model is the *model of expertise* which describes the contents of the knowledge base of the KBS in (essentially) three parts, called *layers*: 1. Inference Layer, specifying the data flow, 2. Task Layer, specifying the control flow, and 3. Domain Layer, specifying the kinds of data. The KADS methodology is widely used for new developments in the field of knowledge based systems. Many knowledge acquisition tools are based on the KADS methodology. It would be interesting to examine, how well GAPCAD meets the requirements of KADS-tools, since

- annotated programs form a logic programming language, which are widely used as prototyping languages for the development of KBS,
- the institute is looking for tools which aid in developing mediatory knowledge bases,
- the Petri-net model is very similar to the description of the inference layer in KADS. This point is discussed in the sequel.

The inference layer of the model of expertise in KADS contains a description of the data flow in the KBS to be developed. Its graphical representation (called *inference structure*) contains *meta-classes* (squares) – the data; and *knowledge sources* or inference steps (circles) – the operations on the data. See Figure 5 as an example for an inference structure. It shows the data flow in a generic configuration task [15]. Notice the similarity to Petri-nets: meta-classes are places and knowledge sources are transitions, but one should notice the dual graphical syntax: circles are squares and vice versa. The semantics of the inference structure is not formally specified in KADS. The syntactical equivalence between the two was used in MoMo [21].

However, as our extended Petri-net model represents specific program clauses on the *symbol level*, the inference structure in KADS describes "only" the very idea, how the expert performs the (configuration) task on the *knowledge level* [14]. Further refinement of

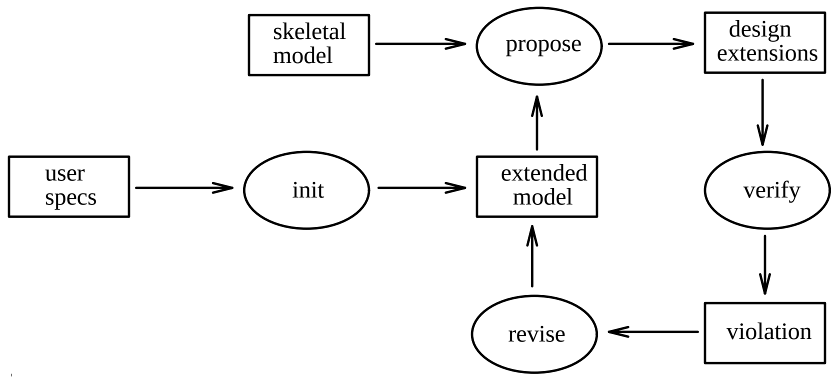


Figure 5: A top level inference structure

the inference structure leads to more special and finally to atomic inference steps. In Figure 6 a more detailed description of the `propose` knowledge source from Figure 5 is shown. Coloured Petri-nets, on which the extended Petri-net model is based, can also be defined as

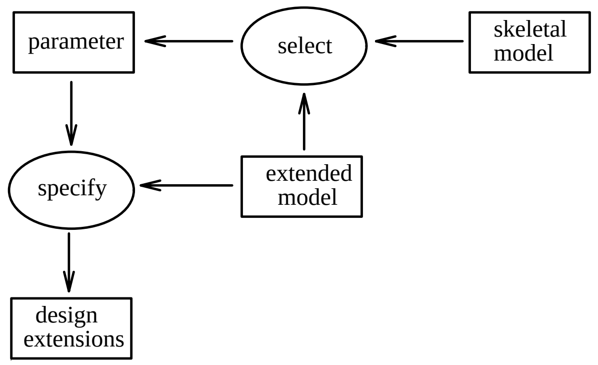


Figure 6: Refinement of inference step 'propose'

hierarchical nets [6]. It would be interesting to investigate the possibility of a graphically and semantically unified top-down-construction of knowledge bases, beginning at the top (knowledge level) with the inference structure similar to Figure 5, applicating some local refinements as in Figure 6, and ending up with an extended Petri-net as a representation for GAPs (symbol level), which can be debugged using GAPCAD and efficiently executed by DAEDALUS. Some related issues need to be addressed:

- Are atomic inference steps really clauses?
- Where do the arc labels fit in? What is their interpretation on the knowledge level?
- How can nets with non-atomic transitions be executed? [11]

6 Conclusion

This report described the theory and the implementation of an executable graphical representation of generalized annotated programs using the notation and semantic of Petri-nets. The following topics were addressed the first time:

- The formalism presented here is based on *Coloured* Petri-nets, which provide a natural way to express constraints on the annotations, using the concept of transition guards. In [2] and [13], predicate/transition nets [4] were used.
- An execution model for the Petri-net based computation of the well-founded model of normal programs was presented. Another alternative is described in [16], which is based on classical logic, rather than on annotated logic.
- Finally the GAPCAD implementation provides an interesting graphical user interface for entering GAPs, and serves as an basis for a more sophisticated tool. Promising ideas in this direction were also presented in this report.

Three possible future research direction are:

- Development of a better Petri-net based realisation of the well-founded semantics.
- Provision of more support for entering GAPs for DAEDALUS (especially for the sorts of the predicates).
- Viewing GAPCAD in a more general frame for support of knowledge acquisition along the lines of [21] and [22].

Acknowledgements I'd like to thank Joachim for supervising and supporting this work, Michael Himsoldt for providing GraphEd and especially Peter for many quick respondings to my never-ending wishes, remarks and misunderstandings concerning DAEDALUS.

References

[1] Bancilhon, F., Maier, D., Sagiv, Y., Ullman, D.D. Magic Sets and other strange ways to implement logic programs *Proceedings of ACM Symposium on Principles of Database Systems, 1986, pp. 1-15*

[2] D. Debertin. Parallizing inference in distributed knowledge based systems. *Master's thesis, Institute of Algorithms and Cognitive Systems, University of Karlsruhe (in German)*

[3] F. Gebhardt, E. Groß, H. Voß. Concurrency constraints as control specifications for the MoMo language *FABEL Report No.21, GMD Sankt Augustin, 1994*

[4] H.J. Genrich. Predicate/Transition nets. *LNCS 254, Springer-Verlag, 1987 pp. 207-247*

[5] M. Hemsoldt. GraphEd User Manual, and Sgraph 3.1 Programmers Manual *Included in GraphEd distribution, available at ftp.uni-passau.de in pub/local/graphed.*

[6] Kurt Jensen. Coloured Petri Nets: A High Level Language for System Design and Analysis. *in: G. Rozenberg (Ed.): Advances in Petri Nets 1990, pp. 342-416*

[7] Michael Kifer, V.S. Subrahmanian. Theory of Generalized Annotated Logic. *Journal of Logic Programming 12, 1992, pp. 335-367*

[8] D.B. Kemp, P.J. Stuckey, D. Scrivastava. Magic Sets and the Bottom-up Evaluation of the Well-founded Model. *Logic Programming: Proceedings 1991 International Symposium, San Diego, pp. 337-351*

[9] Peter Kullmann. SLG-Resolution for Generalized Annotated Logic. *Master's thesis, Institute for Algorithms and Cognitive Systems, University of Karlsruhe, 1995 (in German)*

[10] Jim Lu, Anil Nerode, V.S. Subrahmanian. Towards a Theory of Hybrid Knowledge Bases. *To appear in IEEE Transactions on Knowledge and Data Engineering*

[11] Frank Maurer. Hypermedia-based Knowledge Engineering for Distributed Knowledge Based Systems. *Diss. thesis (in German), infix Sankt Augustin 1993*

[12] P. Meseguer. A new method to checking rule bases for inconsistency: a Petri Net approach. *Proceedings of ECAI, Stockholm, 1990, pp. 437-442*

[13] Tadao Murata, V.S. Subrahmanian, Toshiro Wakayama. A Petri Net Model for Reasoning in the Presence of Inconsistency. *IEEE Transactions on Knowledge and Data Engineering, Vol3, No.3, September 1991, pp.281-292*

[14] Allen Newell. The Knowledge Level. *Artificial Intelligence 18(1982) pp.82-127*

[15] A. Th. Schreiber, P. Terpstra, P. Magni, M. van Velzen. Analysing and Implementing VT Using CommonKADS. *KADS-Workshop.*

[16] T. Shimura, J. Lobo, Tadao Murata. An Extended Petri Net Model for Normal Logic Programs. *IEEE Transactions on Knowledge and Data Engineering, Vol. 7, No. 1, Feb. 1995*

[17] V.S. Subrahmanian. Amalgamating Knowledge Bases. *ACM Transactions on Database Systems 19,2, 1994, pp. 291-331*

[18] V.S. Subrahmanian, S. Adali, A. Brink, R. Emery, Jim Lu, Adil Rajput, T.J. Rogers, R. Ross, C.Ward. HERMES Heterogeneous Reasoning and Mediator System. *Draft, University of Maryland, 1995* Available via WWW.

[19] A. van Gelder, K. Ross and J. Schlipf. The Well-founded Semantics for General Logic Programs. *Journal of the ACM, Vol. 38, No. 3, July 1991, pp. 620-650*

[20] A. van Gelder. The Alternating Fixpoint of Logic Programs with Negation (Extended Abstract) *Proc. 8th Symposium on Principles of Database Systems, March 29-31, Philadelphia 1989*

[21] J. Walther et. al. MoMo *GMD Sankt Augustin 1993, Germany.* Available via WWW

[22] B.J. Wielinga, A.T. Schreiber, J.A. Breuker. KADS: A Modelling Approach to Knowledge Engineering *Knowledge Acquisition, 4:5-53, 1992*

[23] D. Zhang, D. Nguyen. PREPARE: A Tool for Knowledge Base Verification. *IEEE Transactions on Knowledge and Data Engineering, December 1994, Vol. 6, Number 6, pp. 983-989*