Snoopy - A Tool to Design and Animate/Simulate Graph-Based Formalisms

Monika Heiner Ronny Richter Martin Schwarick

Brandenburg University of Technology at Cottbus
Postbox 10 13 44,
Cottbus, Germany
snoopy @ informatik.tu-cottbus.de
http://www-dssz.informatik.tu-cottbus.de/software/snoopy.html

ABSTRACT

We sketch the fundamental properties and features of Snoopy, a tool to model and execute (animate, simulate) hierarchical graph-based system descriptions. The tool comes along with several pre-fabricated graph classes, especially some kind of Petri nets and other related graphs, and facilitates a comfortable integration of further graph classes due to its generic design.

To support an aspect-oriented model engineering, different graph classes may be used simultaneously. Snoopy provides some features (hierarchical nodes, logical nodes), which are particularly useful for larger models, or models with an higher connectivity degree.

There are several Petri net classes available, among them the purely qualitative place/transition nets in its standard definition and in a version enhanced by four special arcs as well as two quantitative extensions of it - stochastic Petri nets and continuous Petri nets. Each of these classes enjoys dedicated animation or simulation features.

Our tool runs on Windows and Linux operating systems, and it is available free of charge for non-commercial use.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*State diagrams, Petri nets*

General Terms

Model Creation and Execution

Keywords

editor, animator, simulator, numerical integration algorithms, qualitative and quantitative Petri nets.

International Workshop on Petri Nets Tools and APplications PNTAP 2008, March 3, 2008, Marseille, France.

1. PRELIMINARIES

The support by tools is a necessary condition for the acceptance of a formalism. The perspective from various abstraction levels by several models of different expressive strength is a crucial point for a sophisticated evaluation of a system under investigation, technical or natural ones.

In this paper, we present Snoopy [35], a generic and adaptive tool for modelling and animating/simulating hierarchical graph-based formalisms. While concentrating our development as far on several kinds of Petri nets and related graph classes, the generic design of Snoopy facilitates also a comfortable extension by new graph classes.

The simultaneous use of several graph classes is supported by the dynamic adaptation of the graphical user interface to the active one. So it is possible to treat qualitative and quantitative models of the system under investigation side by side.

For example you can start with a qualitative Petri net model and increase first your confidence in the net behaviour by animating it, i.e. playing the token game, before checking some essential properties using external analysis tools. Later you can easily move on to related quantitative models, deterministically timed, stochastic or continuous ones, to get a deeper understanding of the time dependencies governing your system. These quantitative models can be simulated using internal or external tools. This integrating approach has been employed in [5], [6]. To support this style of model engineering it is possible to convert different graph classes into each other, obviously with loss of information in some directions.

In the following we use the term *animation* for the visualization of the token game. The token game executes the model qualitatively, according to the defined rules, complemented by non-deterministic choices, if necessary. Contrary, the term *simulation* stands shortly for the numerical evaluation of the studied system as by stochastic or deterministic integration algorithms to solve systems of generally non-linear equations. Even for large models a smooth animation is performed, but the generic data structure of Snoopy induces a lower performance in more expensive simulations compared to dedicated tools. So a wide range of exports to analysis tools is available.

Snoopy runs on Windows and Linux operating systems. It is available free of charge for non-commercial use, and can be obtained from our website [35]. The source code is available on request.

2. GRAPH INDEPENDENT FEATURES

Snoopy provides for all graph classes some consistently available generic features. For example, the graphical editor supplies some fundamental commands like copy, paste and cut, allowing an easy re-use of building blocks. In addition, some advanced layout functions like mirror, flip and rotate as well as an automatic layout by the Graphviz library [31] may be beneficial. A basic printing support and some graphical file export (eps, Xfig, FrameMaker) help for documentation purposes.

The GUI is realized in Windows as MDI-application, which is simulated in Linux by a tabbed document interface. Some arbitrarily placeable mini windows give access to the insertable graph elements and the hierarchy of the model.

All attributes of any graph element, like nodes or edges, may be set not only for single elements, but also for a set of selected elements all at ones.

Graph constraints permit only the creation of syntactically correct models of the implemented graph classes.

The construction of large graphs is supported by a general hierarchy concept of subgraphs (represented as macro nodes) and by logical (fusion) nodes, which serve as connectors of distributed net parts. Additionally, colours or different shapes of individual graph elements may be used to highlight special functional aspects (compare Figure 2).

A generic interaction mode allows a communication between different graphs. Some events in one graph can trigger commands (colouring, creating or deleting of graph elements) in another graph, even if they are instances of different graph classes [3].

Furthermore, a dynamic colouring of graph elements is available to visualize paths or node sets [38]. It is possible to select more than one node set, and to colour the intersection or union of these selected sets.

A digital signature by md5 hash ensures the structure and the layout of the graph separately, which increases the confidence in former analysis results during model development [2].

3. REALIZED GRAPH CLASSES

3.1 Reachability Graph

This simple graph class supports just one node and one arc type, besides comment nodes. The graph nodes can carry a name, a description and a Petri net state, which consists of a list of places and their markings. The arcs may be labelled by arbitrary character strings. For an example see Figure 2, the window in the right lower corner.

3.2 Petri Net

This class of directed bipartite multi-graphs allows qualitative modelling by the standard notion of place/transition Petri nets. An animation by the token game gives first insights into the dynamic behavior and the causality of the model. The token game may be played step-wise or fully automated in forward or backward direction with different firing rules (single, intermediate, or maximal steps). In the automatic mode, encountered dynamic conflicts are resolved randomly.

For modelling of larger systems the concepts of hierarchal nodes and fusion nodes have been proven to be useful. The dynamic colouring of node sets allows to highlight P/T-invariants, structural deadlocks, traps or any other subsets of nodes.

The generic interaction manager [3] permits to construct the reachability graph driven by the token flow animation of the Petri net. Furthermore, the export to a wide range of external analysis tools is available, among them INA, Lola, Maria, MC-Kit, Pep, Prod, Tina (see [34] for tool descriptions) as well as to our own toolbox Charlie [26], [29]. Additionally, an import of a restricted APNN file format supports advanced model sharing with other Petri nets tools.

3.3 Extended Petri Net

This graph class enhances classical place/transition Petri nets by four special arc types: read arcs, reset arcs, equal arcs and inhibitor arcs. Consider the extended Petri net in Figure 1.

- The transition t_0 is connected with p_0 by an *inhibitor arc*. t_0 is enabled, when p_0 has less than two tokens. The firing of t_0 does not change the number of tokens in p_0.
- The transition t_1 is connected with p_0 by a *read arc* and with p_1 by a *reset arc*. t_1 is enabled, when p_0 has at least two tokens. If t_1 fires, then the number of tokens in p_0 will not be changed, but p_1 will become empty.
- The transition t_2 is connected with p_0 by a read arc and an inhibitor arc; and it is connected with p_1 by a reset and a normal arc. t_2 is enabled, when p_0 contains exactly two tokens. If t_2 fires, then the number of tokens in p_0 will not be changed, but p_1 will have five tokens.
- The transition t_3 is connected with p_0 by an *equal arc*. t_3 is enabled, when p_0 has exactly two tokens. If t_3 fires, it consumes these two tokens from p_0.

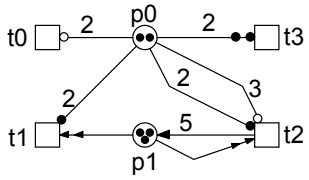

Figure 1: An extended place/transition Petri net, demonstrating the four special arc types.

The token flow animation is available, with all the options as for standard Petri nets, as well as exports to external analysis tools, However, the special arc types are accepted in the export to the APNN file format only, which supports these graph elements.

3.4 Time Petri Net

This class enhances classical place/transition Petri nets by time. Up to now, time durations or time intervals can be assigned to transitions only. The net analysis is supported by an export to the Integrated Net Analyser (INA) [27].

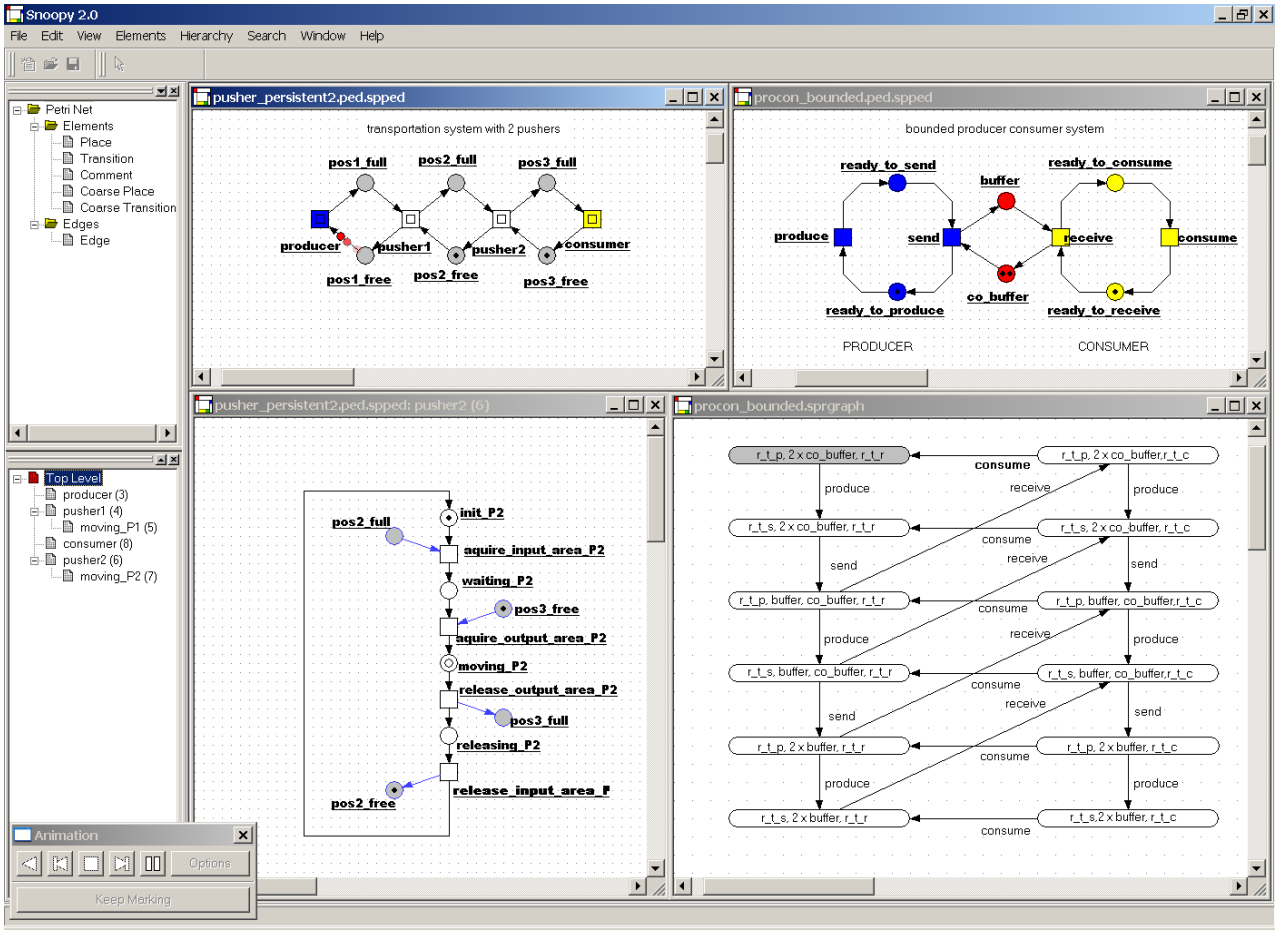

Figure 2: Snoopy Screenshot.

3.5 Stochastic Petri Net

The class of stochastic Petri nets (SPNs) [16] associates a probabilistically distributed firing rate (waiting time) with each transition. Technically, various probability distributions can be chosen to determine the random values for the transitions' local timers.

However, if the firing rates of all transitions follow an exponential distribution, then the behaviour of the stochastic Petri net can be described by a Markovian process. For this purpose, each transition gets its particular, generally marking-dependent parameter λ to specify its rate. The marking-dependent transition rate $\lambda_t(m)$ for the transition t is defined by the stochastic hazard function h_t. The domain of h_t is restricted to the set of pre-places of t, enforcing a close relation between network structure and hazard functions. Therefore $\lambda_t(m)$ actually depends only on a submarking.

To support biochemically interpreted stochastic Petri nets, special types of propensity (hazard) functions are provided, among them the *mass-action propensity function* and the *level propensity function*, see [6] for details.

For illustration we give here one of the most famous examples of mathematical biology - the predator/prey system (Lotka-Volterra system) - as stochastic Petri net, compare Figure 3. It consists of two species, modelled as places: the prey and the predator, and three reactions, modelled as transitions: the reproduction of the prey, the consumption of the prey, and the natural death of the predator.

The three reactions follow the stochastic mass-action kinetics, which basically means that the reaction rates are proportional to the current number of species involved. Having the parameters of the species' interactions (α - reproduction of prey, β - predator death, γ - consumption of prey), we get the following pair of first order, non-linear stochastic equations.

$$\dot{prey} = \alpha \cdot prey - \gamma \cdot prey \cdot predator \quad (1)$$

$$\dot{predator} = \gamma \cdot prey \cdot predator - \beta \cdot predator \quad (2)$$

Applying Gillespie's exact simulation algorithm [7], see Algorithm 1 for a related pseudocode description, produces data as given in the diagrams of Figure 3, describing the dynamic evolution of the biological system over time. Likewise, the results may also be saved in a comma separated value file for further examination by other tools.

Furthermore, two well-established extended stochastic Petri net classes are supported:

- The *generalized stochastic Petri nets* (GSPNs) supply also inhibitor arcs, and immediate transitions.
- The *deterministic and stochastic Petri nets* (DSPNs) provide additionally transitions with deterministic waiting time.

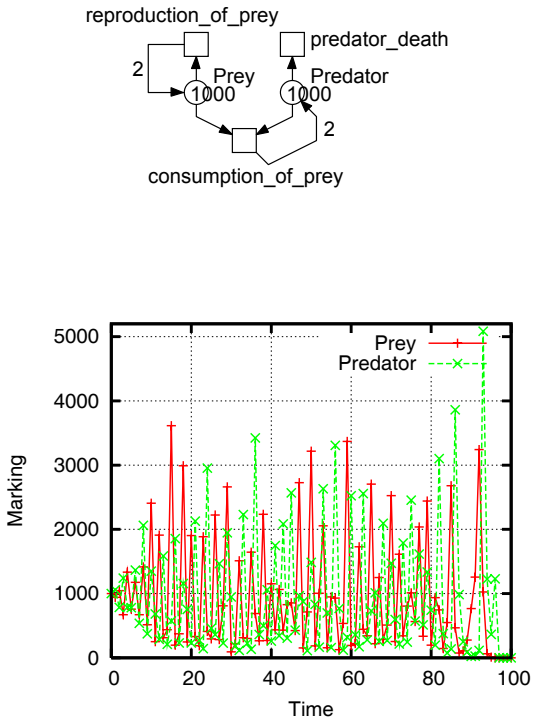

Figure 3: The famous predator/prey example as stochastic Petri net, and its Gillespie simulation.

Thus, DSPNs comprise GSPNs, which in turn comprise SPNs.

Special attention has been paid to the look and feel of the graphical user interface to allow the experimentally working biologist an intuitive and efficient model-based experiment design. In this way, multiple initial markings and multiple parameter sets can be administrated for each model structure [15].

An export to foreign tools providing complementary evaluation techniques is in preparation, as for example to PRISM [20] to allow stochastic model checking, to TimeNet [39] to provide the standard Markovian transient and steady state analysis techniques, or to Dizzy [21] to open access to a wider range of stochastic and deterministic simulation algorithms.

Algorithm 1 Exact Gillespie algorithm for a stochastic Petri net.

given:
SPN with initial marking m_0;
simulation interval $[t_0, t_{max}]$;

time $t := t_0$;
marking $m := m_0$;
$print(t, m)$;
while $t < t_{max}$ **do**
  determine duration τ *until next firing;*
  $t := t + \tau$;
  determine transition tr firing at time t;
  $m := fire(m, tr)$;
  $print(t, m)$
end while

3.6 Continuous Petri Net

In a continuous Petri net [1] the marking of a place is no longer an integer, but a positive real number. Transitions fire continuously, whereby the current deterministic firing rates generally depend on the current marking of the transitions' pre-places, as in the case of stochastic Petri nets. Please note, continuous nodes are drawn in bold lines to distinguish them from the discrete ones, compare Figure 4.

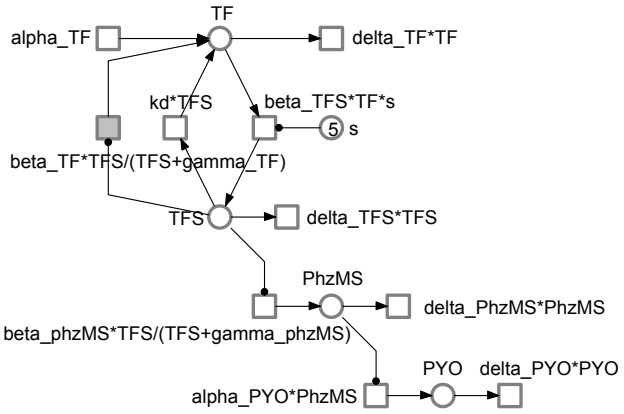

Figure 4: Continuous Petri net(s). The two system versions differ in the transition given in grey, which represents the positive feedback (pfb). For each transition, the continuous rate equation is given.

Continuous Petri nets may be considered as a structured approach to write systems of ordinary differential equations (ODEs), which are commonly used for a quantitative description of biochemical reaction networks (compare section 4). Therefore, some equation patterns like *mass action* and *Michaelis Menten* are supported by our tool [24].

For simulation, a representative set of numerical algorithms is implemented; 12 stiff or unstiff solvers are available, among them Runge-Kutta and Rosenbrock. Their common algorithmic kernel is given in pseudocode notation as Algorithm 2. The results can be displayed on-the-fly in plots or saved in a comma separated value file for further examination by other tools. Our solvers will not compete with other dedicated tools, so we paid more attention on dependability than performance.

Algorithm 2 Basic simulation algorithm for a continuous Petri net.

given:
continuous Petri net with initial marking m_0,
  defining the function f;
simulation interval $[t_0, t_{max}]$;
step size $h < (t_{max} - t_0)$;

time $t := t_0$;
marking $m := m_0$;
$print(t, m)$;
while $t < t_{max}$ **do**
  $t := t + h$;
  $m := m + h \cdot f(m)$;
  $print(t, m)$
end while

There is an export of the continuous Petri net to the Systems Biology Markup Language (SBML) [36] in order to connect to external analysis tools, popular in the systems biology community, as e.g. Copasi [30]. Moreover, the ODEs defined by a continuous Petri net can be generated in LaTeX style for documentation purposes.

Petri nets have been used in the synthetic biology project iGEM [9] to design and construct a completely novel type of self-powering electrochemical biosensor, called ElectrEcoBlu. The novelty lies in the fact that the output signal is an electrochemical mediator, which enables electrical current to be generated in a microbial fuel cell. This was facilitated by the entire team - molecular biologists and engineers/modellers - working in an integrated laboratory environment, using Petri nets as a communication means. The Petri net in Figure 4 generates exactly the ODEs as given in the equations (3) - (6). The last term in equation (3) corresponds to the positive feedback transition (given in grey in Figure 4).

$$\dot{TF} = \alpha_{TF} - \delta_{TF} \cdot TF - \beta_{TFS} \cdot s \cdot TF + k_d \cdot TFS + \beta_{TF} \frac{TFS}{\gamma_{TF} + TFS} \quad (3)$$

$$\dot{TFS} = \beta_{TFS} \cdot s \cdot TF - k_d \cdot TFS - \delta_{TFS} \cdot TFS \quad (4)$$

$$\dot{PhzMS} = \beta_{PhzMS} \frac{TFS}{\gamma_{PhzMS} + TFS} - \delta_{PhzMS} \cdot PhzMS \quad (5)$$

$$\dot{PYO} = \alpha_{PYO} \cdot PhzMS - \delta_{PYO} \cdot PYO \quad (6)$$

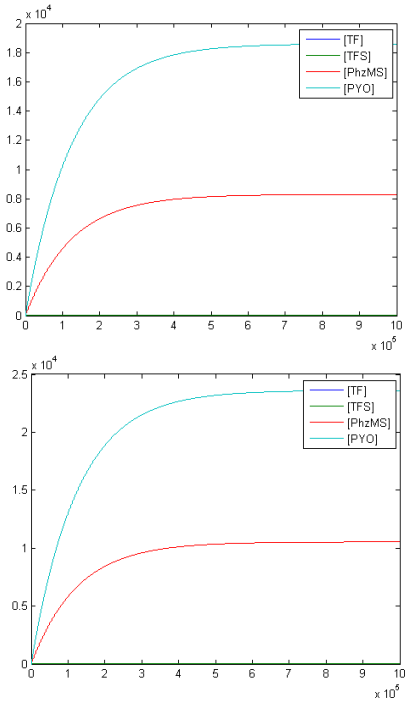

Figure 5: Dynamic behaviour of the continuous Petri net given in Figure 4: (above) simple model, (below) model with feedback. A closer look reveals the speed up by the positive feedback.

Simulating the continuous Petri net, i.e. solving numerically the underlying system of ordinary differential equations, we get diagrams as given in Figure 5.

As in the stochastic case, the administration of multiple initial markings and multiple parameter sets is supported.

3.7 MTBDD

For teaching purposes and documentation of smaller case studies we implemented multi-terminal binary decision diagrams (MTBDD), which obviously comprise the standard notion of binary decision diagrams as special case.

$$M = \begin{pmatrix} 0 & 8 & 0 & 5 \\ 2 & 0 & 0 & 5 \\ 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 \end{pmatrix}$$

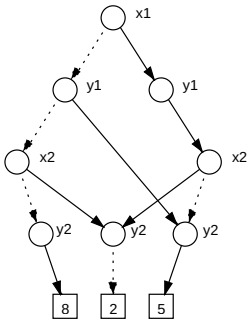

Figure 6: A multi-terminal binary decision diagram (MTBDD) to encode a sparse matrix. The two binary variables x_1, x_2 encode the four row indices, while y_1, y_2 are responsible for the columns.

MTBDDs may be used, for example, to represent sparse matrices, compare Figure 6. Every path to a terminal node encodes the indices of a non-zero entry of a matrix M, the value of which equals the non-terminal node's value. MTBDDs are often exploited to get concise representations of internal data structures, as e.g. in the probabilistic model checker PRISM [20].

3.8 Fault Tree

Fault trees describe the dependencies of component-based systems in failure conditions and are commonly used in risk management of systems with high dependability demands.

In Snoopy are two flavours of fault trees, a basic and an extended class. Both classes confine themselves to one kind of arcs. The following nodes are available in the basic version, compare Figure 7:

- basic event: describes an elementary component failure,
- top event: models the breakdown of the system,
- intermediate event: introduces an internal event, which depend on some basic events,
- coarse event: structures fault trees by hierarchies,
- comment node: for further descriptions,
- AND gate: all input signals must be set to trigger the output,
- OR gate: one input signal must be set to trigger the output,
- NEG gate: the input is negated.

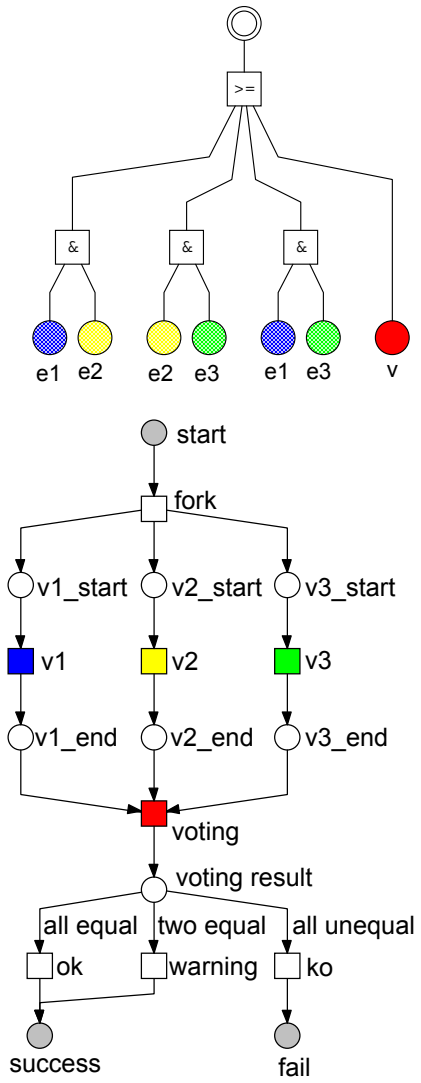

Figure 7: Example of simultaneous use of fault tree and Petri net (N-version programming).

Additionally, the following nodes are available in advanced fault trees, compare Figure 8:

- XOR gate: exactly one input signal must be set to trigger the output,
- m-of-n gate: m of n inputs must be set to trigger the output,
- condition gate: the input must be set and a specified boolean expression has to be true to trigger the output
- condition parameter: defines the boolean expression for the condition gate,
- undeveloped event: defines an event, which is not further considered.

Snoopy provides the qualitative and quantitative evaluation of fault trees. The animation allows a stepwise or automatic visualization of the signal flows resulting in the system breakdown. Moreover, minimal cut sets of basic events, which result into the occurrence of the top event, can be determined. That supports an easy identification of single points of failure.

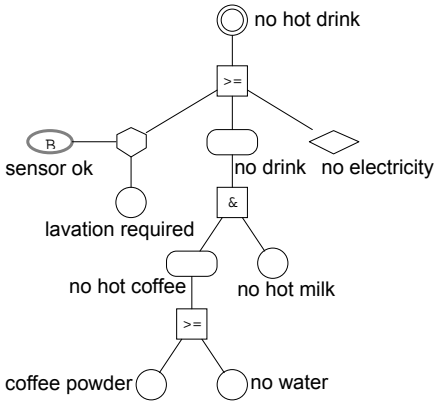

Figure 8: Extended fault tree for a coffee machine.

For a quantitative analysis several dependability measures may be computed for repairable or non-repairable systems. For example the reliability, probability of system failure, availability, mean time to failure, mean time between failures and mean time to repair the system [14] can be computed.

3.9 Miscellaneous

The generic design of our graph tool allows an uncomplicated extension by new graph classes. For example EDL signatures (a formalism to describe patterns of computer network attacks) have been realized in [22]. Snoopy is also involved in the tool chain of the embedded system design approach presented in [8].

You might want to find your own favorite graph class in a future version of this paper - we are open for suggestions and cooperations.

4. CASE STUDIES

Snoopy has been utilized for teaching purposes and students' projects for quite a while. Moreover, it has been used for a wide range of case studies, technical as well as biochemical ones. The following list is not meant to be exhaustive.

4.1 Technical and Academic Case Studies

- concurrent pusher [10]
- control software of a production cell [11]
- solitaire game [28]

4.2 Biological Case Studies

- qualitative models of signal transduction networks: apoptosis [12], haemorrhage [18].
- qualitative models of metabolic networks: glucolysis [23], potato tuber [13].
- qualitative as well as quantitative models of signal transduction networks: RKIP [5], MAPK [6], and extended gene expression networks: biosensor [9].

5. IMPLEMENTATION

5.1 General Information

Snoopy was started in 1997 as a student's project [17], [4] and it is still under development and maintenance. It is based on the experience gathered by its predecessor PED [32], which it replaces.

The tool is written in the programming language C++ using the Standard Template Library. A crucial point of the development is its platform-independent realisation, so Snoopy is now available for Windows and Linux operating systems. A version for Apple/Macintosh is under test. For this purpose, the graphical user interface employs the framework wxWidgets [37].

The object-oriented design uses several standard design patterns (especially Model View Controller, Prototype, and Builder), thus special requirements may be added easily. Due to a strict separation of internal data structures and graphical representation it is straightforward to extend Snoopy by a new graph class applying reuse and specialization of existing elements. As usual, a similar base class has to be selected, inherited elements can be overwritten and new ones can be added from the pool of available templates.

5.2 Internal Data Structures

The main object in the data structure is the graph object which contains methods for modifications and holds the associated node classes and edge classes. Every node class has one prototype and a number of containing nodes that are copies from this prototype. The edge class is similarly structured, as it can be seen in Figure 9. Every node and every edge can have a list of attributes defining the properties of the graph elements.

A graphic is assigned to every displayed element, see Figure 10. Attributes of graph elements may be manipulated by widgets as it is shown in Figure 11. This architecture facilitates the addition of new graph classes in Snoopy.

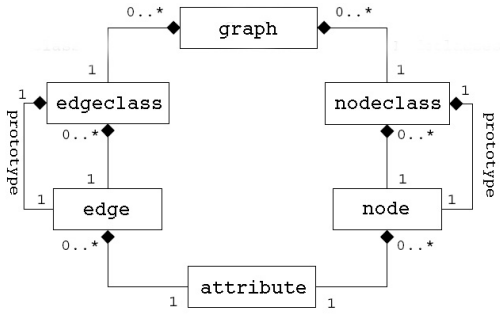

Figure 9: Internal data structure.

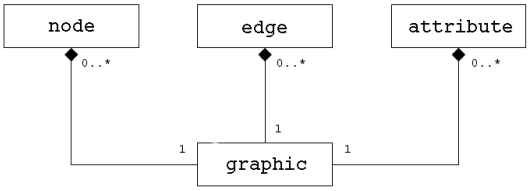

Figure 10: Graphics assigned to the graph elements.

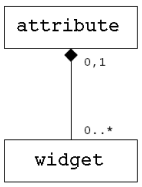

Figure 11: Attributes are connected with window interaction controls.

5.3 Implementation of a new Graph Class

For a better understanding of the following example, we give a short introduction into our naming convention. The name of each implemented class in Snoopy starts with "SP_", followed by one of the prefixes, defining the scope of the class (here a few are given only):

- "DS": element of the data structure,
- "GR": element of the graphics,
- "GRM": event handler for graphical elements,
- "GUI": graphical user interface elements,
- "WDG": window interaction element widget,
- "DLG": window dialog.

We demonstrate the extension of Snoopy by a new graph class called *modulo net*, which has been introduced in [19] to detect and correct operation errors in distributed systems. Modulo nets are basically qualitative place / transition nets extended by undirected arcs and a global integer number P. All adjacent arcs of a place are (exclusively) either conventionally directed arcs or undirected arcs. The firing of a transition changes the marking of a place connected by an undirected arc according to the following rule: $(number\ of\ tokens + arc\ weight) \mod P$; see Figure 12 for an example.

The new graph class must inherit from an existing base class and should overwrite the function "CreateGraph". In listing 1 the class SP_DS_ModuloNets inherits from the base class SP_DS_SimplePed.

In listing 2 the implementation of the method "CreateGraph" calls on line 4 and 5 the homonymous method from SP_DS_SimplePed and inserts afterwards the Modulo Net class's special elements.

Furthermore, a special animator class is assigned to the places (see listing 2, line 14-15). This class calculates the token change according to the semantics of the undirected

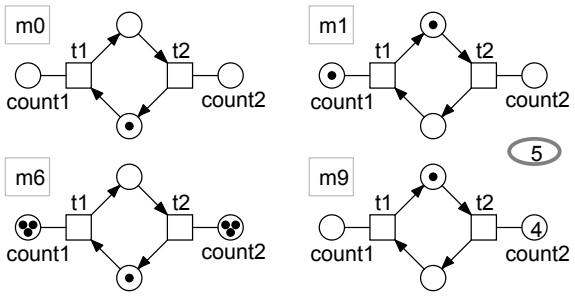

Figure 12: A modulo net and four of its nine markings. The two places *count*1, *count*2 count modulo 5 the number of the transitions' occurrences. Obviously, they can only differ by 1 (modulo 5) in any reachable marking.

arcs (see listing 4). Afterwards the new undirected edge class is added in listing 2, lines 17-24 (please note, arcs are named edges in the implementation). The shape of the arc is defined in listing 2, lines 18-19. In lines 21-24 the attribute multiplicity with its widget and graphical representation is added to the undirected arc.

Finally, the functions "NodeRequirement" and "EdgeRequirement" have to be overwritten to ensure the given constraints over inserted nodes or arcs.

The whole implementation of the new graph class comprises about 350 lines of code and can be done by an experienced user within one day.

Listing 1: Extract of SP_DS_ModuloNets.h

```
class SP_DS_ModuloNets: public                          1
    SP_DS_SimplePed
{                                                       2
  SP_DS_ModuloNets();                                   3
  SP_DS_ModuloNets(const char* p_pchName);              4
                                                        5
  SP_DS_Graph* CreateGraph(SP_DS_Graph*                 6
      p_pcGraph);
                                                        7
  bool NodeRequirement(SP_DS_Node* p_pcNode             8
      );
  bool EdgeRequirement(                                 9
            SP_DS_Edgeclass* p_pcEdgeclass,             10
            SP_Data* p_pcNode1,                         11
            SP_Data* p_pcNode2);                        12
};                                                      13
```

Listing 2: Implementation of CreateGraph

```
SP_DS_Graph*                                            1
SP_DS_ModuloNets::CreateGraph(SP_DS_Graph*              2
    p_graph)
{                                                       3
  if (!SP_DS_SimplePed::CreateGraph(p_graph             4
      )) return NULL;
                                                        5
  SP_DS_Nodeclass* l_pcNodeClass;                       6
  SP_DS_Edgeclass* l_pcEdgeClass;                       7
  SP_DS_Attribute* l_pcAttr;                            8
  SP_Graphic* l_pcGrAttr;                               9
  SP_GR_Node* l_pcGrNode;                               10
                                                        11
  l_pcNodeClass = p_graph->GetNodeclass(                12
      "Place");
  l_pcNodeClass->AddAnimator(new                        13
      SP_DS_ModNetPlAnimator("Marking"));
                                                        14
  l_pcEdgeClass = p_graph->AddEdgeclass(                15
      new SP_DS_Edgeclass(p_graph,
      "Undirected Edge"));
  l_pcGrEdge = new SP_GR_NoArrowEdge(                   16
      l_pcEdgeClass->GetPrototype(),0,0,0);
  l_pcEdgeClass->SetGraphic(l_pcGrEdge);                17
                                                        18
  l_pcAttr = l_pcEdgeClass->AddAttribute(               19
      new SP_DS_MultiplicityAttribute(
      "Multiplicity"));
  l_pcAttr->RegisterDialogWidget(new                    20
      SP_WDG_DialogUnsigned("General", 1));
  l_pcGrAttr = l_pcAttr->AddGraphic(new                 21
      SP_GR_MultiplicityAttribute(l_pcAttr)
      );
  l_pcGrAttr->SetShow(TRUE);                            22
                                                        23
  l_pcNodeClass = p_graph->AddNodeclass(new             24
       SP_DS_Nodeclass(p_graph, "Modulo"));
  l_pcAttr = l_pcNodeClass->AddAttribute(               25
      new SP_DS_NumberAttribute("Modulo",
      1));
  l_pcAttr->RegisterDialogWidget(new                    26
      SP_WDG_DialogUnsigned("General"));
  l_pcGrAttr = l_pcAttr->AddGraphic(new                 27
      SP_GR_TextAttribute(l_pcAttr));
  l_pcGrNode = new                                      28
      SP_GR_ExtendedDoubleParameterSymbol(
      l_pcNodeClass->GetPrototype());
  l_pcNodeClass->SetGraphic(l_pcGrNode);                29
  l_pcNodeClass->RegisterGraphicWidget(new              30
      SP_WDG_DialogGraphic("Graphic"));
return p_graph;                                         31
}// end CreateGraph                                     32
```

Listing 3: Extract of SP_DS_ModNetPlAnimator.h

```
class SP_DS_ModNetPlAnimator: public                    1
    SP_DS_PlaceAnimator
{                                                       2
  private:                                              3
  long m_nModToken;                                     4
  protected:                                            5
  virtual bool DecrementMark();                         6
  long DecrementedMarking(SP_List<                      7
      SP_DS_Edge*>* p_edges, long p_tokens)
      ;
  public:                                               8
  SP_DS_ModNetPlAnimator(const char*                    9
      p_pchAttributeName,
      SP_DS_Animation* p_pcAnim = NULL,                 10
      SP_DS_Node* p_pcParent = NULL);                   11
  virtual ~SP_DS_ModNetPlAnimator();                    12
};                                                      13
```

Listing 4: Implementation of DecrementedMarking

```
long SP_DS_ModNetPlAnimator::
    DecrementedMarking(SP_List<SP_DS_Edge
    *>* p_edges, long p_tokens)
{
  if (!p_edges) return p_tokens;
  long l_nOldVal = m_pcAttribute->GetValue
      ();
  SP_DS_Graph* l_pcGraph = SP_Core::
      Instance()->GetRootDocument()->
      GetGraph();
  SP_DS_Nodeclass* l_pcNodeclass =
      l_pcGraph->GetNodeclass("Modulo");
  SP_DS_Node* l_pcNode = static_cast<
      SP_DS_Node*> (* l_pcNodeclass->
      GetElements()->begin());
  wxString l_sModuloVal = wxT(l_pcNode->
      GetAttribute("Modulo")->
      GetValueString());
  long l_nModuloVal;
  l_sModuloVal.ToLong(&l_nModuloVal);

  SP_DS_Attribute* l_pcAttr;
  SP_List<SP_DS_Edge*>::iterator l_eIt;

  for (l_eIt = p_edges->begin(); l_eIt !=
      p_edges->end(); ++l_eIt)
  {
    if (strcmp((*l_eIt)->GetClassName(),
        "Undirected Edge") == 0 )
    {
    l_pcAttr = (*l_eIt)->
        GetFirstAttributeByType(
        SP_ATTRIBUTE_MULTIPLICITY);
    if(l_pcAttr)
    {
      long l_nArcMulti = static_cast<
          SP_DS_MultiplicityAttribute*> (
          l_pcAttr)->GetValue();
      m_nModToken += (l_nOldVal +
          l_nArcMulti ) % l_nModuloVal;
    }//end if
    }//end if
  }//end for

return m_nModToken;
}//end DecrementedMarking
```

6. FUTURE WORK

The import of biochemical network models from KEGG and SBML data formats is about to be released [25], which will allow the direct re-use and re-engineering of models from the systems biology community.

There is a well-known (syntactically) close relation between biochemically interpreted stochastic and continuous Petri nets. Obviously, an automatic conversions of the stochastic and continuous rate functions could be of help for the investigation of related biochemical network models.

Up to now, we consider pure net classes only, meaning all nodes have to be either discrete or continuous ones. We need hybrid nets to integrate both aspects into one model. Hybrid models might help to investigate the interrelations between the discrete and continuous parts of a network.

An ongoing student's project aims at managing and executing animation sequences, especially counter examples produced by external analysis tools. Another projects investigates automatic coarsening of network structures.

Finally, a PNML [33] import and export will be implemented as soon as a (preliminary) final standard will be available.

7. ACKNOWLEDGEMENT

We wish to acknowledge Thomas Menzel for his initial contributions to the development of Snoopy, and Alexey Tovchigrechko for his valuable enhancements, as well as the work done by former students, including Matthias Dube, Markus Fieber, Anja Kurth, Sebastian Lehrack, Christian Rohr, Daniel Scheibler, and Katja Winder. We also acknowledge financial support by MPI Martinsried and MPI Magdeburg in developing the stochastic component.

8. REFERENCES

[1] R. David and H. Alla. *Discrete, Continuous, and Hybrid Petri Nets*. Springer, 2005.

[2] M. Dube. Signing and Verifying SNOOPY files. Technical Report, Università degli Studi di Milano, Dipartimento di Informatica e Comunicazione, 2005.

[3] M. Dube. Design and Implementation of a Generic Concept for an Interaction of Graphs in Snoopy (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2007.

[4] M. Fieber. Design and Implementation of a Generic and Adaptive Tool for Graph Manipulation (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2004.

[5] D. Gilbert and M. Heiner. From Petri Nets to Differential Equations - an Integrative Approach for Biochemical Network Analysis;. In *Proc. 27th ICATPN 2006, LNCS 4024*, pages 181–200. Springer, 2006.

[6] D. Gilbert, M. Heiner, and S. Lehrack. A Unifying Framework for Modelling and Analysing Biochemical Pathways Using Petri Nets. In *Proc. CMSB 2007, LNCS/LNBI 4695*, pages 200–216. Springer, 2007.

[7] D. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[8] L. Gomes, J. Barros, and A. Costa. Petri Nets Tools and Embedded Systems Design. In *Proc. Workshop on Petri Nets and Software Engineering (PNSE07) at Int. Conf. on Application and Theory of Petri Nets (ICATPN '07 Siedlce)*, pages 214–219, 2007.

[9] C. Harkness. The Use of Petri Nets in the Glasgow iGEM Project: ElectrEcoBlu – a Self-powering Electrochemical Biosensor. Technical report, University of Glasgow, Bioinformatics Research Centre, 2007.

[10] M. Heiner. Verification and Optimization of Control Programs by Petri Nets without State Explosion. In *Proc. 2nd Int. Workshop on Manufacturing and Petri Nets at Int. Conf. on Application and Theory of Petri Nets (ICATPN '97 Toulouse)*, pages 69–84, 1997.

[11] M. Heiner, P. Deussen, and J. Spranger. A Case Study in Design and Verification of Manufacturing Systems

with Hierarchical Petri Nets. *Journal of Advanced Manufacturing Technology*, 15:139–152, 1999.

[12] M. Heiner, I. Koch, and J. Will. Model Validation of Biological Pathways Using Petri Nets - Demonstrated for Apoptosis. *BioSystems*, 75:15–28, 2004.

[13] I. Koch, B. H. Junker, and M. Heiner. Application of Petri Net Theory for Modeling and Validation of the Sucrose Breakdown Pathway in the Potato Tuber. *Bioinformatics*, 21(7):1219–1226, 2005.

[14] A. Kurth. Fault Trees in Snoopy (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2007.

[15] S. Lehrack. A Tool to Model and Simulate Stochastic Petri Nets in the Context of Biochemical Networks (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2007.

[16] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing, John Wiley and Sons, 1995. 2nd Edition.

[17] T. Menzel. Design and Implementation of a Framework for Petri Net Oriented Modelling (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 1997.

[18] G. Neumann. Modeling of Biochemical Processes with Petri Nets; Hemostasis vs. Fibrinolysis vs. Inhibitors (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2004.

[19] A. Pagnoni. Detecting and Correcting Operation Errors of Distributed Systems. *Bulletin of the EATCS*, 58, 1996.

[20] Parker, D.A. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, School of Computer Science, 2002.

[21] S. Ramsey. Dizzy User Manual. Technical report, Institute for Systems Biology, CompBio Group, Seattle, Washington, 2006.

[22] C. Rohr. Design and Implementation of an Editor for Visualizing and Debugging of EDL Signatures (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2007.

[23] T. Runge. Model Engineering and Validation of Biochemical Networks with Coloured Petri nets - Demonstrated for Glycolysis (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2004.

[24] D. Scheibler. A Tool to Design and Simulate Continuous Petri Nets (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2006.

[25] D. Schrödter. Re-engineering of Biochemical Networks (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., submitted, 2007.

[26] M. Schwarick. A Tool to Analyze Petri Nets (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2006.

[27] P. Starke and S. Roch. INA - The Intergrated Net Analyzer; User Manual. Technical report, Humboldt University Berlin, www.informatik.hu-berlin.de/∼starke/ina.html, 1999.

[28] P. H. Starke. Halma net. *Petri Net Newsletter*, 28:3–8, 1987.

[29] Website Charlie. A Tool for the Analysis of Place/Transition Nets. `http://www-dssz.informatik.tu-cottbus.de/software/charlie/charlie.html`.

[30] Website COPASI. Complex Pathway Simulator. `http://www.copasi.org`, last visit: 07/2007.

[31] Website Graphviz. Graph Visualization Software. `http://www.graphviz.org` last visit: 08/2007.

[32] Website PED. An Hierarchical Petri Net Editor. `http://www-dssz.informatik.tu-cottbus.de/software/ped.html`.

[33] Website Petri Net Markup Language. PNML Framework Release 1.2.0. `http://www-src.lip6.fr/logiciels/mars/PNML/`, last visit: 08/2007.

[34] Website Petri Nets World. The Home Site. `http://www.informatik.uni-hamburg.de/TGI/PetriNets/`, last visit: 08/2007.

[35] Website Snoopy. A Tool to Design and Animate/Simulate Graphs. `http://www-dssz.informatik.tu-cottbus.de/software/snoopy.html`.

[36] Website Systems Biology Markup Language. The Home Site. `http://sbml.org/index.psp`, last visit: 08/2007.

[37] Website wxWidgets. A Toolkit for cross-platform GUI Application. `http://www.wxwidgets.org`, last visit: 08/2007.

[38] K. Winder. Invariant-based Structural Characterization of Petri Nets (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2006.

[39] A. Zimmermann and M. Knole. TimeNET 4.0: A Software Tool for the Performability Evaluation with Stochastic and Coloured Petri Nets: User Manual. Technical report, Technical University Berlin, Real-Time Systems and Robotics Group, TR 2007-13, 2007.