PATOOL – A TOOL FOR DESIGN AND ANALYSIS OF DISCRETE SYSTEMS USING PROCESS ALGEBRAS WITH FDT INTEGRATION SUPPORT

Slavomír ŠIMOŇÁK, Ivan PEŤKO
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics,
Technical University of Košice, Letná 9, 042 00 Košice, Slovak Republic, tel.: +421 55 602 4220,
e-mail: slavomir.simonak@tuke.sk; ivan.petko@toryconsulting.sk

ABSTRACT
This paper deals with PATool – a process algebra tool with FDT integration support. We discuss main features, used formats, principles of conversions and cooperation with other tools in order to provide a platform for the design and analysis of systems using process algebras. PATool is based on former work in the FDT integration area, which formed the basic idea and requirements to the tool. As a result, PATool currently offers a simple interface for FDT integration tools and process algebra specifications with corresponding interchange formats transformations. A typical use is demonstrated by examples.

Keywords: *process algebras, ACP, APC, CSP, Petri nets, formal methods*

1. INTRODUCTION

One of the many available definitions states that formal description techniques (FDT or simply formal methods) are formally sound (mathematically-based) techniques for the specification, development and verification of software and hardware systems. The principal aim of FDT usage is to enable designers to construct real-life size systems that work correctly with respect to requirements.

Process algebras [2], [3], [14], [15] are very significant part of FDT offering algebraic methods for description and analysis of discrete systems. Many tools were developed to support work with process algebras [5] and the majority of them support the specific process algebra only. In order to integrate process algebras and other FDT a number of FDT integration techniques and tools were developed (e.g., integration of process algebras and Petri nets [6]). Reasons mentioned above led us to the idea to develop a tool, with a support for various process algebras and some additional formal methods as well.

In this paper, we introduce the PATool – a tool for process algebras including also some FDT integration tools support. Since many significant tools exist already there, each of them supporting one specific process algebra, it's not necessary to develop similar tool. Instead, a different approach is used in PATool development. Starting from FDT integration tools supported (ACP2PETRI [9] and PETRI2APC [8]) it is not hard to see that there is a need for support of process algebras ACP (Algebra of Communicating Processes) [2] and APC (Algebra of Process Components) [4].

The main contribution of PATool to a process of system description and analysis is its ability to convert formats used by external tools. PATool thus represents an interface to multiple process algebra notations. Considering a built-in interface between ACP/APC and CSP (Communicating Sequential Processes) [3], very interesting analysis techniques implemented in external tools (e.g., CWB-NC [5]) are available now for analysis of ACP and APC specifications. By this means, PATool takes the advantage of great intellectual work and years of active research devoted to development of such tools.

2. PATOOL OVERVIEW

PATool (Process Algebras Tool) principles were firstly proposed in [1] in order to give a formal background for the tool supporting process algebras and FDT integration. The primary motivation was the absence of a tool which supports multiple process algebras and formats of existing FDT integration tools. There are many tools developed to this day, each supporting the only one specific formalism (for instance CWB/CCS (Calculus of Communicating Systems) [10], FDR2/CSP [11], PSF Toolkit/ACP [12], PEPA Tools/PEPA (Performance Evaluation Process Algebra) [13], etc.) so PATool has not an ambition to offer similar properties for more process algebras. Instead of it, PATool cooperates with existing FDT integration tools by means of conversions of their formats, offers process algebras transformations in order to use significant properties of professional external tools, provides an interface between their representations and integrates formalisms used by means of format conversions and thus allows the user to use specifications written in any of supported notations.

Currently PATool provides standard text editing functionalities, (i.e., load, save, new, cut, copy…) and format conversions, supporting the following file formats:

- CSP format – textual representation of the CSP algebra, used by CWB-NC [5],
- ACP textual and PAML (Process Algebras Markup Language) format – textual and PAML representation of ACP specifications,
- APC textual and PAML format – textual and PAML representation of APC specifications.

Cooperating independent FDT integration and external analysis tools mentioned above, PATool interacts with, are ACP2PETRI [9], PETRI2APC [8] and CWB-NC [5]. PATool also provides a direct execution of external tools with converted specifications. It also provides a GUI functionality to originally non-GUI tools with a built-in specification syntax check. To provide the interface between cooperating tools and user specifications, there are four types of conversions available:

- ACP in textual format to ACP in PAML format
- ACP in PAML (an input of ACP2PETRI) format to ACP in textual format
- APC in PAML (an output of PETRI2APC) format to APC in textual format
- ACP and APC textual and PAML formats to CSP format.

3. PATOOL PRINCIPLES

In the following, PATool principles are considered briefly in order to provide main ideas behind the tool. Since the tool is based on some of the results obtained in FDT integration area, we refer to [1], [4], [6], [8] and [9] for a formal background. In [1] format definitions and conversions are described in details (PAML formats are defined by DTD specifications and textual formats are defined by formal grammars).

3.1. Interfaces

The main interface which PATool provides is an interface between the user and specifications in ACP (textual or PAML), APC (textual or PAML) and CSP (textual) - note that CSP specifications can be obtained from those written in ACP or APC (both textual or PAML form). This interface can also be viewed as three separate sub-interfaces (Fig. 1).

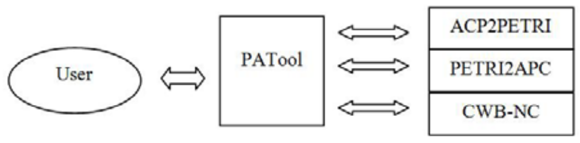


Fig. 1 PATool interface

Since the ACP2PETRI input format is an ACP PAML (XML-based) specification, PATool allows to define the specification in much simpler (textual) mode, i.e., it provides the conversion from ACP textual format to ACP PAML format and vice versa. User may interactively specify and modify the specification, run ACP2PETRI directly from PATool and evaluate its output.

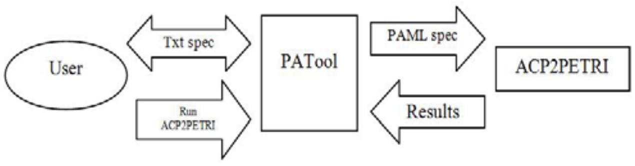


Fig. 2 PATool/ACP2PETRI interface

Note that the output of ACP2PETRI is the PNML (Petri Net Markup Language) specification of corresponding Petri Net. PATool displays the tool output, so the user can see the ACP2PETRI messages generated during the processing.

Unlike the ACP2PETRI tool, PETRI2APC takes the PNML specification as an input and generates the APC specification in PAML format. This is quite complicated to read and therefore PATool converts the specification generated to APC textual format, which is much simpler to understand. It is possible to run the PETRI2APC directly from PATool, (PETRI2APC generated messages are displayed), automatically load the APC PAML output and convert it to textual form. Thus, the whole activity can be done again from the PATool environment.

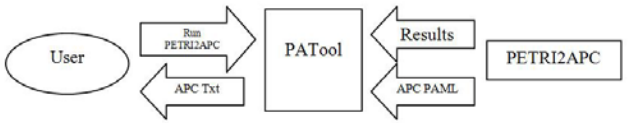


Fig. 3 PATool/PETRI2APC interface

Once the user has obtained or created ACP or APC specification in PAML or textual format (as an output of PETRI2APC or written by the user in order to analyze it), or has written the CSP specification directly, he is able to analyze it by an external tool using the conversion into the CWB-NC's CSP notation (or, essentially, by any tool which supports the CWB-NC's general syntax for CSP algebra). Every ACP or APC specification in both textual and PAML format can be converted into the corresponding CSP specification (with some restrictions given in [1]) using PATool and thus analysis of a specification in any of the formats supported is possible.

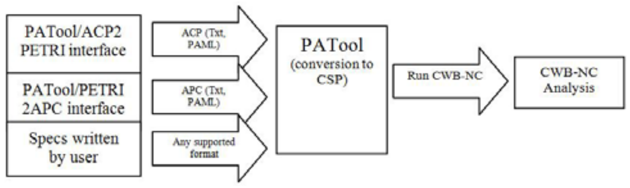


Fig. 4 PATool/CWB-NC interface (analysis)

3.2. Used formats

Each file format PATool supports has its own predefined structure depending on the specific type. Basically formats supported can be subdivided into two types – textual and XML-based. Predefined PATool file formats [1] are acp (file with a textual representation of ACP specification), apc (file with a textual representation of APC specification), paml (XML file with respect to given DTD specification with representation of ACP or APC algebra).

The textual format of ACP algebra [1] was defined by the formal grammar [7] in order to provide a simple mechanism for creating (writing) ACP specifications. A syntax checking feature is also provided to support writing ACP specifications. Such a specification can be stored in a file with any extension, although .acp is preferred.

Similarly, the APC textual representation is also defined by the formal grammar to simplify readability of APC specifications. The ACP and APC PAML formats [1] given by the particular DTD specifications allow to store specifications equivalent to those in textual form, but more suitable for machine processing.

The used formats are defined as follows.

Def.: Let G_{ACP} be a context-free grammar

$G_{ACP} = (N_{ACP}, T_{ACP}, P_{ACP}, start)$, where

N_{ACP} = {*start, gamma, encset, acpequation, act, var, apcterm, acpeq, A, B, C, D, actions, encaps, actionSet, actt, id*} is a set of non-terminal symbols, where *start* is a starting non-terminal symbol.

T_{ACP} = {'gamma', '(', ')', ',' ,'=', 'encset', 'encaps', '[', ']', '+', '||', '.', 'delta', 'epsilon' } is a set of terminal symbols.

P_{ACP} is a set of grammar rules in form $A \rightarrow \beta$ (written in extended Backus-Naur form), where A is a non-terminal symbol and β is a regular expression over the set of terminal and non-terminal symbols, defined as:

P_{ACP}:
start $\rightarrow$ gamma encset acpequation
gamma $\rightarrow$ ['gamma' '(' act ',' act ')' '=' act gamma]
encset $\rightarrow$ ['encset' '[' id ']' '(' actionSet ')' encset]
acpequation $\rightarrow$ var '=' acpterm acpeq
acpeq $\rightarrow$ [acpequation]
acpterm $\rightarrow$ A
A $\rightarrow$ B['+' A]
B $\rightarrow$ C['||' B]
C $\rightarrow$ D['.' C]
D $\rightarrow$ encaps | '(' A ')' | act | var
act $\rightarrow$ (a| .. |z)$^{+}$(0| ... |9)*(a| .. |z|A| ...|Z|_)* | 'delta' | 'epsilon'
actionSet $\rightarrow$ [act actions]
actions $\rightarrow$ [',' actt]
actt $\rightarrow$ act actions
var $\rightarrow$ (A| .. |Z)$^{+}$(0| ... |9)*(a| .. |z|A| ...|Z|_)*
encaps $\rightarrow$ 'encaps' '[' id ']' '(' acpterm ')'
id $\rightarrow$ act | var

Notice that the non-terminals *act* and *var* are defined by regular expressions [7]. The rules in the set P_{ACP} preserve the following precedence convention:

Def.: Let p: $O \rightarrow N$ be a precedence function, where O is a set of ACP operators and N is a set of natural numbers. For ACP algebra we have

$$O = \{+, ., ||, \partial_H(), (\ )\},$$

where + is alternative composition, . is sequential composition, || is parallel composition, $\partial_H()$ is encapsulation and ( ) stands for bracketing. Then

$$p(+) < p(||) < p(.) < p((\ )) = p(\partial_H())$$

In our grammar, +, . and || are identical to their syntactic equivalents in ACP, *encset[id]* is a definition of encapsulation set, i.e., *encaps[id]* stands for $\partial_{id}()$, *gamma* represents a binary communication function γ.

Similarly we define the APC textual form.
Def.: Let G_{APC} be a context-free grammar

$$G_{APC} = (N_{APC}, T_{APC}, P_{APC}, start), \text{ where}$$

N_{APC} = {*start, pidef, apcequation, componentList, process, var, proc, componentStart, componentEnd, apcterm, comp, eqcomponent, eq, apceq, A, B, C, D, act, id*} is a set of non-terminal symbols, and *start* is a starting non-terminal symbol.

T_{APC} = {'pi', '(', ')', ',','=', '[', ']', '+', '||', '.', 'delta', 'epsilon', '#', '{', '}'} is a set of terminal symbols.

P_{APC}:
start $\rightarrow$ pidef apcequation
pidef $\rightarrow$ ['pi' '(' componentList ')' '=' process pidef]
process $\rightarrow$ proc | var '=' proc
proc $\rightarrow$ apcterm
componentList $\rightarrow$ componentStart apcterm componentEnd ',' componentStart apcterm componentEnd comp
comp $\rightarrow$ [',' componentStart apcterm componentEnd comp]
apcequation $\rightarrow$ eqcomponent apceq | eq apceq
eq $\rightarrow$ var '=' apcterm
eqcomponent $\rightarrow$ componentStart eq componentEnd
apceq $\rightarrow$ [apcequation]
apcterm $\rightarrow$ A
A $\rightarrow$ B['+' A]
B $\rightarrow$ C['||' B]
C$\rightarrow$ D['.' C]
D $\rightarrow$ '(' A ')' | act | var | componentStart A componentEnd
componentStart $\rightarrow$ '#' '[' id ']' '{'
componentEnd $\rightarrow$ '}'
act $\rightarrow$ (a| .. |z)$^{+}$(0| ... |9)*(a| .. |z|A| ...|Z|_)*| 'delta' | 'epsilon'
var $\rightarrow$ (A| .. |Z)$^{+}$(0| ... |9)*(a| .. |z|A| ...|Z|_)*
id $\rightarrow$ act|var

The APC precedence function is defined such that

$$p(+) < p(||) < p(.) < p((\ ))$$

Notice, that the ACP and APC grammars are very simple deterministic context-free grammars of type LL(1).

In order to support PAML specifications, the following DTD definitions representing XML tree structures are defined:

ACP PAML format:
```
   <!-- ACP DTD for process specifications -->
   <!ELEMENT ACPSPEC
(GAMMA*,ENCSET*,ACPEQUATION+)>
   <!ELEMENT ACPEQUATION (VAR,ACPTERM)>
   <!ATTLIST ACPEQUATION INIT CDATA #REQUIRED>
   <!ELEMENT ACPTERM
(ALTCMP|SEQCMP|PARCMP|ACTION|VAR|ENCAPS)>
   <!ELEMENT ALTCMP
(ALTCMP|SEQCMP|PARCMP|ACTION|VAR|ENCAPS)+>
   <!ELEMENT SEQCMP
(ALTCMP|SEQCMP|PARCMP|ACTION|VAR|ENCAPS)+>
   <!ELEMENT PARCMP
(ALTCMP|SEQCMP|PARCMP|ACTION|VAR|ENCAPS)+>
   <!ELEMENT ENCAPS
(ALTCMP|SEQCMP|PARCMP|ACTION|VAR|ENCAPS)>
   <!ATTLIST ENCAPS ENCID CDATA #REQUIRED>
   <!ELEMENT ACTION EMPTY>
   <!ATTLIST ACTION NAME CDATA #REQUIRED>
   <!ELEMENT VAR EMPTY>
   <!ATTLIST VAR NAME CDATA #REQUIRED>
   <!ELEMENT GAMMA EMPTY>
   <!ATTLIST GAMMA ACT1 CDATA #REQUIRED ACT2
CDATA #REQUIRED RES CDATA #REQUIRED>
   <!ELEMENT ENCSET (ACTION*)>
   <!ATTLIST ENCSET ENCID CDATA #REQUIRED>
```

APC PAML format:

```
   <!-- APC DTD for process specifications -->
   <!ELEMENT APCSPEC
(PIDEFINITION*,APCEQUATION+)>
   <!ELEMENT APCEQUATION (VAR,APCTERM)>
   <!ATTLIST APCEQUATION INIT CDATA #REQUIRED
PROC CDATA #IMPLIED >
   <!ELEMENT APCTERM
(ALTCMP|SEQCMP|PARCMP|ACTION|VAR)>
   <!ATTLIST APCTERM PROC CDATA #IMPLIED >
   <!ELEMENT ALTCMP
(ALTCMP|SEQCMP|PARCMP|ACTION|VAR)+>
   <!ATTLIST ALTCMP PROC CDATA #IMPLIED >
   <!ELEMENT SEQCMP
(ALTCMP|SEQCMP|PARCMP|ACTION|VAR)+>
   <!ATTLIST SEQCMP PROC CDATA #IMPLIED >
   <!ELEMENT PARCMP
(ALTCMP|SEQCMP|PARCMP|ACTION|VAR)+>
   <!ATTLIST PARCMP PROC CDATA #IMPLIED >
   <!ELEMENT ACTION EMPTY>
   <!ATTLIST ACTION NAME CDATA #REQUIRED PROC
CDATA #IMPLIED>
   <!ELEMENT VAR EMPTY>
   <!ATTLIST VAR NAME CDATA #REQUIRED PROC CDATA
#IMPLIED>
   <!ELEMENT PIDEFINITION
(COMPONENT,COMPONENT+,RESULT) >
   <!ATTLIST PIDEFINITION ID CDATA #REQUIRED >
   <!ELEMENT COMPONENT (APCEQUATION|APCTERM) >
<!ELEMENT RESULT (APCEQUATION|APCTERM) >
```

Since the definitions are intuitive and we suppose that the reader is familiar with DTD and XML concepts, we consider them as clear enough. Notice that process and action names and other syntactical information are determined by attributes of the specifications.

3.3. Format conversion principles

Since the ACP and APC textual representations are given by the formal grammars [7], the conversion to their PAML equivalent is trivial with respect to the formal translation theory. As noticed above, the ACP and APC grammars are very simple deterministic context-free grammars of type LL(1), i.e. a top-down parser reading an input from left using the only one symbol to determine the next grammar rule to use may be implemented. In fact, PATool translation algorithms implement the SDTS (Syntax Directed Translation Scheme) concept [7], which is one of possible translation principles of the translation theory, i.e. for each input terminal symbol there exists an output terminal symbol. In the real implementation each non-terminal symbol is represented by a method and each terminal symbol is represented by the call of lexical analyzer procedure returning the terminal. When a particular input terminal symbol is read from the source, a corresponding output symbol is written to the output. This leads to the SDTS. After having a look at the relevant grammars for ACP or APC, it is not hard to imagine the conversion principle (the rules of relevant grammar are rewritten into the source code by the principles of translation theory – as described above). In our approach each terminal is „packed up” to the corresponding PAML element (i.e. the PAML elements are output symbols of a translation grammar extending the $G_{APC/ACP}$ grammar with a set of output symbols and translation rules of the same form as the $G_{APC/ACP}$ ones provided that for each terminal symbol the rules are extended with corresponding PAML symbols as an output), before entering a method for a given non-terminal symbol the starting element is written and at the moment of returning from the method the ending element is written to the output (that is why „syntax directed”). Let us illustrate the principle on a small example. The ACP rule

encaps $\rightarrow$ *'encaps' '[' id ']' '(' acpterm ')'*

is rewritten into the code:

```
     private String encaps(){

        String start, end, encaps;
        start = end = encaps = "";

        symbol = getSymbol();
        if (!checkSymbol(symbol,
ACPTxtSymbol.SYMBOL_ENCAPS)) syntaxError(...);

        symbol = getSymbol();
        if (!checkSymbol(symbol,
ACPTxtSymbol.SYMBOL_LPAR_ID)) syntaxError(...);

        String id = id();

        //write the corresponding PAML tag,
//i.e., write the corresponding output symbol
        start = writeStartPAMLTag(..., id, ... );

        symbol = getSymbol();
        if (!checkSymbol(symbol,
ACPTxtSymbol.SYMBOL_PPAR_ID))syntaxError(...);
        symbol = this.acpTxtSymbol.getSymbol();
        if (!checkSymbol(symbol,
ACPTxtSymbol.SYMBOL_LPAR)) syntaxError(...);

        encaps = acpterm();
        symbol = getSymbol();
        if (!checkSymbol(symbol,
ACPTxtSymbol.SYMBOL_PPAR))syntaxError(...);

        //write the corresponding PAML tag
        end = writeEndPAMLTag(...);

        return start + encaps + end;
    }
```

When converting from PAML format to its textual representation, PATool at first gets the tree structure of the source (remark that PAML is a special case of XML given by the DTD specification) and from this structure it is very easy to obtain relevant textual symbols by processing nodes of the structure. As in the previous case, for each PAML element there exists a terminal symbol of a textual representation and it is clear that each node represents some expression (given by sub-nodes).

In both PAML to text and text to PAML conversion, there must be the operator precedence respected by the translation. SDTS respects this implicitly due to the operator precedence given by grammar rules. Processing the PAML tree structure must take care of the operator precedence using the other way. For operator nodes, action nodes and variables nodes, if the sub-node represents the node with higher precedence as the „parent” node, there is no conflict. If the precedence of sub-node is lower than the precedence of parent node, the expression represented by the whole sub-node must be put into the parenthesis to make sure that the precedence will be preserved also in the textual equivalent of PAML being translated. There is an example of the tree structure of a PAML specification depicted in Fig. 5. In such a tree structure, internal nodes represent operators and control elements, whereas leaves are variables and actions.

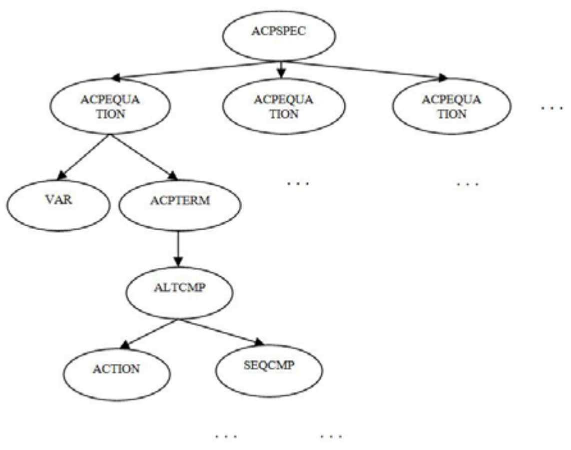


Fig. 5 Tree structure of a PAML specification

Since each node has explicitly defined its acceptable child nodes (see the DTD specification), the only thing to do is to write the corresponding output symbol when processing a particular node (for each node a method is implemented), e.g., the ALTCMP element definition

```
   <!ELEMENT ALTCMP
(ALTCMP|SEQCMP|PARCMP|ACTION|VAR|ENCAPS)+>
```

may be rewritten into the following (abstract) source code:

```
   processAltCmp(){
        childNodes = getChildNodes();

        for each childNode in childNodes {
                switch(childNode.type) {
                case ACTION:
                        write(childNode.value);
                case VAR:
                        write(childNode.value);
                case ALTCMP:
                        processAltCmp();
                case SEQCMP:
                        processSeqCmp();
                case PARCMP:
                        processSeqCmp();
                case ENCAPS:
                        processEncaps();
                }
                if (childNode.last = FALSE)
                        write( „ + “) ;

        }       //end of for each cycle
   }    //end of method implementation
```

The conversion of APC and APC PAML formats to CSP format requires the conversion of PAML to textual format using the principles described above. The conversion of ACP and APC textual formats to the CSP is described deeply in [1]. This conversion takes an advantage of similarity between ACP and CSP algebra, and (since ACP and APC are closely related) also the APC and CSP similarity (very similar operational behavior). To be more exact, ACP and APC specifications are „simulated” in CSP.

The conversions of ACP and APC specifications to CSP specification take an advantage of similarities between operators (in fact their similar operational behaviour and axioms). Firstly we identified closely related operators (e.g., CSP external choice ≈ ACP choice, CSP interleaving and synchronization ≈ ACP parallel composition) and used mechanisms such as temporal renaming and synchronization to simulate the effect desired. In order to preserve the operator precedence we used the parenthesis (ACP, APC and CSP operator precedence is different). The conversions are proposed in a way, where resulting CSP term is written directly in the CWB-NC-like syntax. This should not be an obstacle for using a different syntax, since conversion principles are general enough and after a slight modification only, the translation can be used for any CSP syntax with the base CSP operators supported.

The conversion of ACP to CSP is given by the following formal translation function

$T_*: TERM_{ACP} \rightarrow TERM_{CSP}$

with its subfunctions:

$T_{act}: TERM_{ACP} \rightarrow TERM_{CSP}$	[action translation]
$T_+: TERM_{ACP} \rightarrow TERM_{CSP}$	[choice translation]
$T_.: TERM_{ACP} \rightarrow TERM_{CSP}$	[seq. translation]
$T_{()}: TERM_{ACP} \rightarrow TERM_{CSP}$	[parenthesis expr.]
$T_{\|\|,\|\|\_}: TERM_{ACP} \rightarrow TERM_{CSP}$	[interleaving trans.]
$T_{\|}: TERM_{ACP} \rightarrow TERM_{CSP}$	[communication trans.]
$T_{\partial}: TERM_{ACP} \rightarrow TERM_{CSP}$	[encapsulation]

The definitions of the main translation function and its subfunctions (in the following we suppose that the reader is familiar with the CSP syntax) are as follows:

$T_{act}[t] = a\ , \forall a \in ACTIONS \wedge\ a \notin last(t) \wedge a \in t$
$T_{act}[t] = a \rightarrow SKIP, \forall a \in ACTIONS \wedge a \in last(t)$,
otherwise a term t could be considered as a deadlocked one by a CSP analyzer iff $a \in last(t)$ and the term does not end with the $SKIP$ constant
$T_{act}[t] = X, \forall X \in VARS \wedge\ X \in t$
$T_{act}[\delta] = STOP$ (deadlock)
$T_{act}[\varepsilon] = SKIP$ (no process, but not deadlock)
$T_{act}[t] = t$, otherwise

$T_+[t] = T_*[x]\ []\ T_*[y]$, iff $t = x + y$ ($[]$ = external choice, i.e., a choice influenced by an environment)
$T_+[t] = t$, otherwise

$T_.[t] = T_*[x]; T_*[y]$, iff $t = x.y \wedge (x \notin ACTIONS \vee\ x = \delta \vee\ x = \varepsilon)$
$T_.[t] =\ x \rightarrow T_*[y]$, iff $x \in ACTIONS \wedge\ t = x.y \wedge x \neq \varepsilon \wedge x \neq \delta \wedge t = x.y$
$T_.[t] = t$, otherwise

$T_{()}[t] = (T_*[x])\ ,\ x \in TERM_{ACP} \wedge\ t = (x)$
$T_{()}[t] = t$, otherwise

$T_{||,||\_}\ [t] = (((T_*[x])\ |||\ (T_*[y]))\ []\ ((T_*[x][[a_1 <\!\!-\ c_1, b_1 <\!\!-\ c_1, \dots\ a_n <\!\!-\ c_n, b_n <\!\!-\ c_n\ ]])\ [|\{c_1, \dots, c_n\}|]\ (T_*[y]\ [[a_1 <\!\!-\ c_1, b_1 <\!\!-\ c_1, \dots\ a_n <\!\!-\ c_n, b_n <\!\!-\ c_n]])))$, iff $t = x||y \wedge \gamma(a_1, b_1) = c_1, \dots , \gamma(a_n, b_n) = c_n$ i.e., communication function γ is defined for $a_1, \dots, a_n$ and $b_1, \dots, b_n$ (x contains $a_1, \dots, a_n$ and y contains $b_1, \dots, b_n$). The parenthesis are used to ensure the correct precedence since it may be different in ACP and CSP algebras.

$T_{||,||\_}[t] = ((T_*[x])\ |||\ (T_*[y]))$, iff $t = x||y$ and γ is not defined for actions contained in x or y, i.e., a resulting term is an interleaving.

$T_{||,||\_}[\varepsilon\ ||\_\ x] = STOP$

$T_{||,||\_}[t] = first(ax) \rightarrow (T_{||,||\_}\ [x||y])$, iff $t = a.x||\_y \wedge a = first(a.x) \in ACTIONS \wedge x = REMOVE(first(a.x), a.x)$, where $REMOVE(a, z)$ removes the first occurrence of a symbol a from a string z.

$T_{||,||\_}[t] = (first(x) \rightarrow (T_{||,||\_}\ [x'||z]))\ []\ (first(y) \rightarrow ((T_{||,||\_}\ [y'||z])))$, iff $t = (x + y)||\_z \wedge x' = REMOVE(first(x), x) \wedge y' = REMOVE(first(y), y)$

$T_{||,||\_}[t] = STOP$, iff $t = \varepsilon|x \vee t=x|\varepsilon$

$T_{||,||\_}[t] = t$, otherwise (i.e., t does not contain $||\_$ or $||$)

$T_|[a|b] = (T_*[a]\ [[a <\text{-}\ c]]\ [|\{c\}|]\ (T_*[b][[b <\text{-}\ c]])), \forall a,b \in ACTIONS$, iff $\gamma(a,b) = \mathrm{c}$

$T_|[a|b] = STOP, \forall a,b \in ACTIONS$, iff $\gamma(a,b)$ is not defined.

$T_|[x|y] = (((T_*[x]\ [[A]])\ [|\{C\}|]\ (T_*[y]\ [[B]])))$, iff $t = x|y$, $A = B = \{a_1 <\text{-}\ c_1, b_1 <\text{-}\ c_1, \dots a_n <\text{-}\ c_n, b_n <\text{-}\ c_n\}$ and $C = \{c_1, \dots, c_n\}$ iff $\gamma(a_1, b_1) = c_1, \dots , \gamma(a_n, b_n) = c_n$ i.e., γ communication function is defined for $a_1, \dots, a_n$ and $b_1, \dots, b_n$ (x contains $a_1, \dots, a_n$ and y contains $b_1, \dots, b_n$), $A = B = C = \emptyset$ iff γ is not defined for actions contained in x, y.

$T_|[x|(y + z)] = (T_|[x|y])\ []\ (T_|[x|z])$
$T_|[(x + y)|z] = (T_|[x|z])\ []\ (T_|[y|z])$
$T_|[t] = t$, otherwise

$T_\partial[t] = ((T^*(x))\ [|\alpha x|]\ (\mu.X(a_1 \rightarrow X\ []\ \dots\ []\ a_n \rightarrow X)))$, iff $t = \partial_{\mathrm{H}}(x) \wedge \alpha x - H = \{a_1, \dots, a_n\}$, where μ is the CSP recursion operator and αx is an alphabet of a process x
$T_\partial[t] = STOP$, iff $t = \partial_H(x) \wedge \alpha x - H = \emptyset$
$T_\partial[t] = SKIP$, iff $t = \partial_H(\varepsilon)$
$T_\partial[t] = t$, otherwise

The main function is defined as a composition of the subfunctions, thus

$T_*[t] = T_{()}[t] \circ T_|[t] \circ T_.[t] \circ T_{||,||\_}[t] \circ T_+[t] \circ T_{act}[t] \circ T_\partial[t] = T_\partial[T_{act}[T_+[T_{||,||\_}[T_.[T_|[T_{()}[t]]]]]]]$, for $t \in TERM_{ACP}$

It is not hard to see that the ACP communication is simulated by synchronization of actions (with temporal renaming in corresponding terms in order to allow the actions to be synchronized). Notice that the subfunctions of the translation function are also defined explicitly for some special terms (e.g., $a.x||\_y$, $x|(y + z)$…) in order to preserve their semantics. This respects the operational behaviour of the terms in ACP being simulated by the translation to CSP.

The APC to CSP translation takes advantage of similarity between the ACP and APC definition. Let us define the translation function

$T_{APC*}: TERM_{APC} \rightarrow TERM_{CSP}$

with subfunctions

$T_{act}, T_+, T_., T_{()}, T_{||,||\_}, T_{|||}: TERM_{APC} \rightarrow TERM_{CSP}$

where $T_{act}, T_+, T_., T_{()}$ are defined similarly like in the ACP case with respect to the domain of APC terms (which in fact is the same as in the case of ACP, except of process components).

If process components are not included in APC specification, then $T_{||,||\_}$ is defined as follows:

$T_{||,||\_}[t] = ((T_*[x])\ |||\ (T_*[y]))$, iff $t = x||y$
$T_{||,||\_}[t] = first(x) \rightarrow ((T_*[x'])\ |||\ (T_*[y]))$, iff $t = x||\_y \wedge |first(x)| = 1 \wedge x' = REMOVE(first(x), x)$
$T_{||,||\_}[t] = (first(x_1) \rightarrow ((T_*[x_1'])\ |||\ (T_*[y]))\ []\ \dots\ []\ first(x_n) \rightarrow ((T_*[x_n'])\ |||\ (T_*[y])))$, iff $t = x||\_y \wedge |first(x)| = n$, where $first(x) = \{first(x_1), \dots, first(x_n)\} \wedge x_i' = REMOVE(first(x_i), x_i)$, i.e., $x = (x_1 + \dots + x_n)$
$T_{||,||\_}[t] = t$, otherwise, i.e., t does not contain $||$ nor $||\_$ and process components.

If process components are included in APC specification, then $T_{||,||\_}$ is slightly modified in order to ensure the „non-executability“ of process components involved by their definition (it follows from the APC definition that process components are „executable“ if and only if they are joined together with respect to the π composition function and the join defines a full process, i.e., $\pi(c_1, c_2, \dots, c_n) = p \wedge c_i \in P_C \wedge p \in P$, where P_C is a set of process components and P is a set of processes). The translation of process components exploits the fact, that their form is the same as the form of a process defined by π composition. Remark that if $|||$ is used in APC terms below, then $|||$ represents π composition, if $|||$ is used in CSP terms, the meaning is interleaving (as in CSP terms above).

$T_{||,||\_}[t] = (((T_{APC*}[x])\ [|\{C \cup Z\}|]\ (T_{APC*}[y]))[[*\_componentAction <\text{-}*, C\_componentAction <\text{-}C\ ]])$, iff $t = x||y \vee t = x\ |||\ y \wedge c = \pi(c_1, c_2) \wedge Z =$ set of actions of the components c_1, c_2 (renamed from the from $act\_componentAction$ to act) $\wedge$ x contains c_1 and y contains c_2. By $*\_componentAction<\text{-}*$ we mean renaming of all actions of components c_1 and c_2 (removing the „*_componentAction*“ suffix). C is a set of all actions, which are included in both x and y and renamed from the form *act*_componentAction to *act*. This renaming is given by $C\_componentAction<\text{-}C$. The translation may be extended to the case of more components contained in x and y in a very intuitive way
$T_{||,||\_}[t] = ((T_{APC*}[x])\ [|\{\}|]\ (T_{APC*}[y]))$, iff $t = x||y \vee t = x\ |||\ y \wedge \pi$ is not defined or x and y do not contain the components, for which π is defined.
$T_{||,||\_}[t] = first(x) \rightarrow (T_{APC*}[x'||y])$, iff $|first(x)| = 1$, $t = x||\_y \wedge x' = REMOVE(first(x), x)$
$T_{||,||\_}[t] = (first(x_1) \rightarrow (T_{APC*}[x_1'||y])\ []\ \dots\ []\ first(x_n) \rightarrow (T_{APC*}[x_n'||y]))$, iff $t = x||\_y \wedge |first(x)| = n$, where $first(x) = \{first(x_1), \dots, first(x_n)\} \wedge x_i' = REMOVE(first(x_i), x_i)$, i.e., $x = (x_1 + \dots + x_n)$
$T_{||,||\_}[t] = t$, otherwise, i.e., t does not contain $||$ or $||\_$

$T_{|||}[x|||y] = T_{||,||\_}[x\ |||\ y]\ \ \forall \mathrm{x,y} \in \mathrm{TERM_{APC}}$
$T_{|||}[t] = t$, other (i.e., t does not contain $|||$)

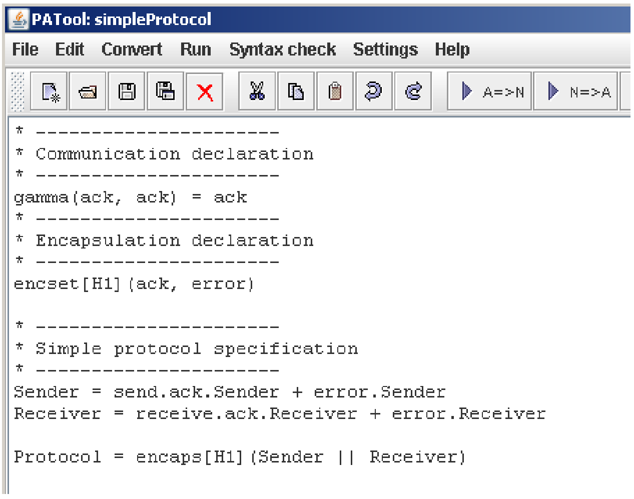
The main function is defined as a composition of the subfunctions, thus:

$$T_{APC^*}[t] = T_0[t] \circ T_{|||}[t] \circ T_{.}[t] \circ T_{||,||\_}[t] \circ T_{+}[t] \circ T_{act}[t]$$
$$= T_{act}[T_{+}[T_{||,||\_}[T.[T_{|||}\,[T_0[t]]]]]], \text{ for } t \in TERM_{APC}$$

Since CSP does not contain a concept of non-executing actions and the only valid actions are those, which do not have to be combined together to execute (in contrast to actions of process components in ACP), there is no way to simulate this behaviour at the syntactical level. This justifies the approach chosen to denote such actions with the „*_componentAction*" suffix to signal that such an action is not executable itself. At the semantic level it is possible to define logical expressions to recognize actions with the suffix as deadlocked ones for CSP analyzer.

4. EXAMPLE

In this section we briefly demonstrate some of the PATool abilities of format conversions which allow the FDT integration and specification analysis. Suppose we have created the ACP specification (Fig. 6) of a simple protocol. In order to analyze the specification we have two possible ways how to use the PATool – either translate it into a Petri net (PN) and take advantages of PN analysis or translate it into a CSP specification and exploit a CSP analyzer for an algebraic analysis. PN or CSP specifications are the outputs of PATool and these specifications may be subsequently analyzed using a convenient external tool, e.g. the CWB-NC tool integrated with PATool.

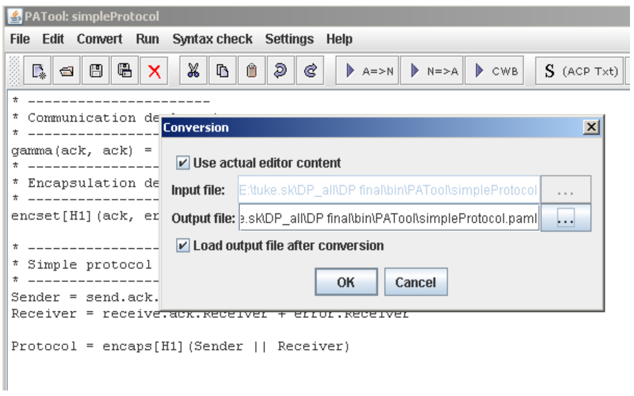


```
* -----------------------
* Communication declaration
* -----------------------
gamma(ack, ack) = ack
* -----------------------
* Encapsulation declaration
* -----------------------
encset[H1](ack, error)

* -----------------------
* Simple protocol specification
* -----------------------
Sender = send.ack.Sender + error.Sender
Receiver = receive.ack.Receiver + error.Receiver

Protocol = encaps[H1](Sender || Receiver)
```

Fig. 6 ACP textual specification

Let us firstly convert the specification into a Petri net - to this end we convert the specification (Fig. 7) to a PN using its corresponding PAML format (Fig. 8) as an intermediate step, which is used by the ACP2PETRI tool (note the simplicity of original specification compared to the equivalent PAML specification).

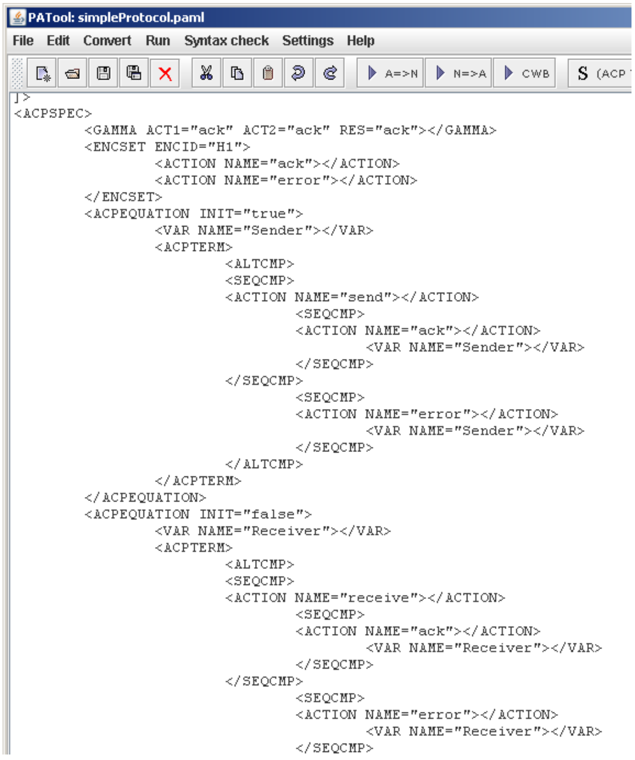
PATool provides the file to file or the editor content to file conversion. This functionality is allowed due to user's possible modifications to the file originally loaded. When converting to PAML, the top-down parsing is used, since the specification is given by the deterministic grammar G_{ACP} of type LL(1) and each input terminal symbol is wrapped into its corresponding PAML element (Fig. 8).



```
* -----------------------
* Communication de
* -----------------
gamma(ack, ack) =
* -----------------
* Encapsulation de
* -----------------
encset[H1](ack, er

* -----------------
* Simple protocol
* -----------------
Sender = send.ack.
Receiver = receive.ack.Receiver + error.Receiver

Protocol = encaps[H1](Sender || Receiver)
```

[✓] Use actual editor content
Input file: E:\tuke.sk\DP_all\DP final\bin\PATool\simpleProtocol
Output file: e.sk\DP_all\DP final\bin\PATool\simpleProtocol.paml
[✓] Load output file after conversion

Fig. 7 Conversion of ACP textual to PAML format

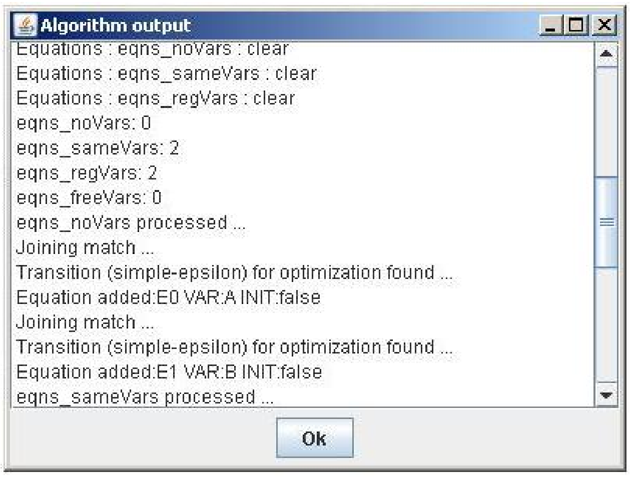

PATool: simpleProtocol.paml
File Edit Convert Run Syntax check Settings Help
A=>N
N=>A
CWB
S (ACP

```
]>
<ACPSPEC>
        <GAMMA ACT1="ack" ACT2="ack" RES="ack"></GAMMA>
        <ENCSET ENCID="H1">
                <ACTION NAME="ack"></ACTION>
                <ACTION NAME="error"></ACTION>
        </ENCSET>
        <ACPEQUATION INIT="true">
                <VAR NAME="Sender"></VAR>
                <ACPTERM>
                        <ALTCMP>
                        <SEQCMP>
                        <ACTION NAME="send"></ACTION>
                                <SEQCMP>
                                <ACTION NAME="ack"></ACTION>
                                        <VAR NAME="Sender"></VAR>
                                </SEQCMP>
                        </SEQCMP>
                                <SEQCMP>
                                <ACTION NAME="error"></ACTION>
                                        <VAR NAME="Sender"></VAR>
                                </SEQCMP>
                        </ALTCMP>
                </ACPTERM>
        </ACPEQUATION>
        <ACPEQUATION INIT="false">
                <VAR NAME="Receiver"></VAR>
                <ACPTERM>
                        <ALTCMP>
                        <SEQCMP>
                        <ACTION NAME="receive"></ACTION>
                                <SEQCMP>
                                <ACTION NAME="ack"></ACTION>
                                        <VAR NAME="Receiver"></VAR>
                                </SEQCMP>
                        </SEQCMP>
                                <SEQCMP>
                                <ACTION NAME="error"></ACTION>
                                        <VAR NAME="Receiver"></VAR>
                                </SEQCMP>
```

Fig. 8 Resulting specification in the PAML format

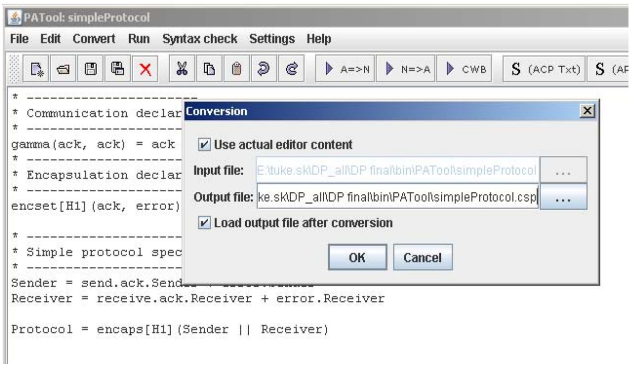
After the conversion, the ACP2PETRI tool can be started directly from the PATool environment (Fig. 9), with the PAML specification supplied (Fig. 8) as an input producing a PN semantically corresponding to the original ACP specification.

After the ACP2PETRI tool processing has been performed (Fig. 9), the output is a PN in the PNML format (the output file generated by the ACP2PETRI tool) corresponding to the original ACP textual specification and thus we are able to analyse the specification with some of available Petri net tools. If an analysis by algebraic means is preferred (the second way of analysis of the original ACP specification), PATool allows to transform the specification (in textual or PAML format) into its corresponding CSP notation (Fig. 11) using the formal translation function *T*[]* and use an external tool subsequently (integrated CWB-NC or any other tool supporting the CWB-NC-like syntax of CSP specifications).



```
Equations : eqns_noVars : clear
Equations : eqns_sameVars : clear
Equations : eqns_regVars : clear
eqns_noVars: 0
eqns_sameVars: 2
eqns_regVars: 2
eqns_freeVars: 0
eqns_noVars processed ...
Joining match ...
Transition (simple-epsilon) for optimization found ...
Equation added:E0 VAR:A INIT:false
Joining match ...
Transition (simple-epsilon) for optimization found ...
Equation added:E1 VAR:B INIT:false
eqns_sameVars processed ...
```

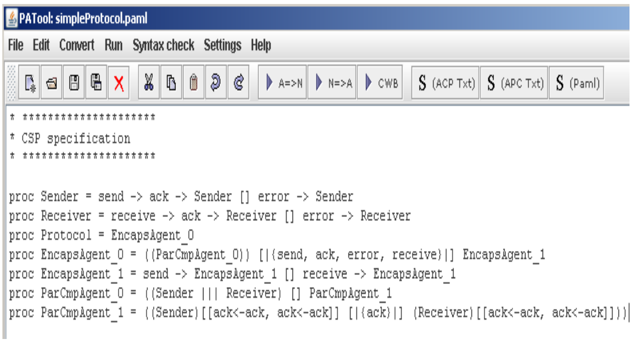
Fig. 9 ACP2PETRI tool processing



```
* ---------------------
* Communication declar
* --------------------
gamma(ack, ack) = ack
* --------------------
* Encapsulation declar
* --------------------
encset[H1](ack, error)

* --------------------
* Simple protocol spec
* --------------------
Sender = send.ack.Send
Receiver = receive.ack.Receiver + error.Receiver

Protocol = encaps[H1](Sender || Receiver)
```

☑ Use actual editor content
Input file: E:\tuke.sk\DP_all\DP final\bin\PATool\simpleProtocol
Output file: ke.sk\DP_all\DP final\bin\PATool\simpleProtocol.csp
☑ Load output file after conversion

Fig. 10 Conversion of ACP specification to CSP


PATool: simpleProtocol.paml
File Edit Convert Run Syntax check Settings Help
A=>N N=>A CWB (ACP Txt) (APC Txt) (Paml)

```
* ********************
* CSP specification
* ********************

proc Sender = send -> ack -> Sender [] error -> Sender
proc Receiver = receive -> ack -> Receiver [] error -> Receiver
proc Protocol = EncapsAgent_0
proc EncapsAgent_0 = ((ParCmpAgent_0)) [|{send, ack, error, receive}|] EncapsAgent_1
proc EncapsAgent_1 = send -> EncapsAgent_1 [] receive -> EncapsAgent_1
proc ParCmpAgent_0 = ((Sender ||| Receiver) [] ParCmpAgent_1
proc ParCmpAgent_1 = ((Sender)[[ack<-ack, ack<-ack]] [|{ack}|] (Receiver)[[ack<-ack, ack<-ack]]))
```

Fig. 11 Resulting CSP specification

5. CONCLUSIONS

In this paper we described the PATool (a process algebra tool with FDT integration support) briefly. The main contribution of the tool to system modeling and formal analysis is the multiple process algebras support allowing the user for writing specifications and cooperating with other FDT integration tools, converting these specifications to appropriate formats and providing it by one unifying interface. We also briefly discussed tool's main functionalities by describing its user interfaces, supported format conversions and principles of integration. The typical usage of the tool is demonstrated by examples. We did not discuss some relevant topics related to PATool – such as cooperating tools and their principles. These may be found in [5, 8, 9].

Since the tool is under constant development, there are many ideas for further improvements. Currently, PATool does not support any of ACP or APC analyzers and simulators (an issue which is solved partially by converting such a specification to CSP and using the external tool). Some minor „aesthetical" improvements are also proposed, such as dividing the tool's functionalities into independent threads or coloring the keywords of specifications. The PATool has an ambition to integrate significant process algebras and offer functionalities, which are not offered by other tools and provide the unifying interface. There is also an ambition to support more FDT integration tools (as they are developed).

REFERENCES

[1] PEŤKO, I.: The environment for design and analysis of discrete systems based on formal methods, Diploma thesis, Košice, Technical University of Košice, Faculty of electrical engineering and informatics, 2008, 104 pages (in Slovak)

[2] BAETEN, J.C.M. - WEIJLAND, W.P.: Process Algebra. Cambridge University Press, 1990, 247 pages, ISBN 0521400430

[3] HOARE, C.A.R.: Communicating Sequential Processes [online], 2004, 238 pages, http://www.usingcsp.com/cspbook.pdf

[4] ŠIMOŇÁK, S. - HUDÁK, Š. - KOREČKO, Š.: APC Semantics for Petri Nets, *Informatica, Volume 32,* 3, 2008, pp. 253-260, ISSN 1854-3871, ISSN 0350-5596, http://www.informatica.si/

[5] The Concurrency Workbench of New Century (CWB-NC) [online], Department of Computer Science, SUNY at Stony Brook, 2000, http://www.cs.sunysb.edu/~cwb/

[6] ŠIMOŇÁK, S.: FDT integration using transformation of Petri nets and process algebras, Dissertation thesis, Košice, Technical University of Košice, Faculty of Electrical Engineering and Informatics, 2003, 112 pages (in Slovak)

[7] KOLLÁR, J. - HAVLICE, Z.: Technology of language systems, Košice, Technical University of Košice, Faculty of electrical engineering and informatics, 2004, 183 pages. ISBN 80-89066-12-7 (in Slovak)

[8] ŠIMOŇÁK, S. - HUDÁK, Š. - KOREČKO, Š.: PETRI2APC: towards unifying Petri nets with other

formal methods, *Proceedings of the Seventh International Scientific Conference Electronic Computers and Informatics ECI 2006, Košice - Herľany, Slovakia*, September 20-22, 2006, Vienala press, 2006, Editors: Štefan Hudák, Ján Kollár, pp. 140-144, ISBN 80-8073-598-0

[9] ŠIMOŇÁK, S. - HUDÁK, Š. - KOREČKO, Š.: ACP2PETRI: a tool for FDT integration support, Analele Universitatii din Oradea, *Proc. 8th International Conference on Engineering of Modern Electric Systems, Felix Spa*, 26.-28. 5. 2005, University of Oradea, Romania, 2005, pp. 122-127, ISSN 1223-2106

[10] Edinburgh Concurrency Workbench, http://homepages.inf.ed.ac.uk/perdita/cwb/

[11] Formal Systems (Europe) Ltd, http://www.fsel.com/

[12] PSF - Process Specification Formalism, Universiteit van Amsterdam, Faculty of Science, http://staff.science.uva.nl/~psf/

[13] PEPA - Performance Evaluation Process Algebra, Laboratory for Foundations of Computer Science, University of Edinburgh, http://www.dcs.ed.ac.uk/pepa/

[14] TOMÁŠEK, M.: Controlling Communication and Mobility by Types with Behavioral Scheme. Acta Polytechnica Hungarica, Vol. 5, No. 4, ISSN 1785-8860, pp. 29 – 40, Budapest, 2008

[15] TOMÁŠEK, M.: Behavioral Scheme of Mobile Processes. Journal of Information Control and Management Systems, Vol. 5, No. 2, ISSN 1336-1716, pp. 371 – 382, Žilina, 2007

Received July 6, 2009, accepted January 22, 2010

BIOGRAPHIES

Slavomír Šimoňák was born in 1974. He graduated in computers and informatics from the Technical University of Košice in 1998. He received his PhD degree in the field of computer devices and systems in 2004. At present he is an assistant professor at Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice. His scientific interests are oriented towards formal methods for design and analysis of discrete systems and their integration, problems related to theory of programming and machine-oriented languages.

Ivan Peťko was born in 1984. He graduated in 2008 with honours at Technical University of Košice in computers and informatics. Currently he is a PhD student at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics at Technical University of Košice. His scientific research is focused on formal description techniques, especially Petri nets and their de/compositional analysis.