SNexpression: A Symbolic Calculator for Symmetric Net Expressions

Lorenzo Capra[1], Massimiliano De Pierro[2](✉), and Giuliana Franceschinis[3]

[1] Dip. di Informatica, Università di Milano, Milan, Italy
lorenzo.capra@unimi.it

[2] Dip. di Informatica, Università di Torino, Turin, Italy
massimiliano.depierro@unito.it

[3] DISIT, Università del Piemonte Orientale, Alessandria, Italy
giuliana.franceschinis@uniupo.it

Abstract. The paper presents SNexpression: a tool for the symbolic structural analysis of Symmetric Nets (SN). It can operate at a low level, handling expressions required to compute the structural properties of interest, but features also a net-based way of interaction allowing to submit commands referring directly to the net structure avoiding error prone input of low level expressions. The User Interface implements a command line interpreter and provides also a multi-page notebook to keep track of the submitted commands and their result.

1 Introduction

The SNexpression tool has been developed with the aim of providing support to the structural analysis of Symmetric Nets (SN), a High-Level Petri Net (HLPN) formalism, without *unfolding* the net, allowing one to work at symbolic and parametric[1] level. A recently added feature is the possibility of deriving a set of *symbolic* ordinary differential equations (Symbolic ODE - SODE) from a Stochastic SN (SSN) model, making it possible an efficient computation of the average marking of colored tokens into places. A first version of SNexpression was presented in [6], but significant improvements/extensions have been implemented since then.

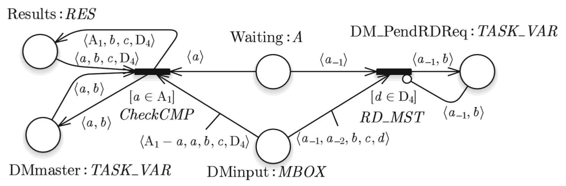
The theoretical work behind the tool has been published in a few papers defining the language for expressing the structural relations in symbolic form and the operators to be applied to the SN arc functions to derive several structural relations [5,7] or to generate a set of SODE from a model satisfying certain properties [3,4]. The basic idea consists of using a syntax similar to the SN arc expressions one, to symbolically represent structural relations useful for checking invariance properties, to deduce model behavioral properties, etc. Each symbolic structural relation is representative of several structural relations defined on the model unfolding: the latter can be derived from the former by instantiating it on specific colors. This approach has advantages: the compact representation,

[1] The method is parametric in the size of the *color classes*.

the similarity of the languages used to describe the model and that used to express the structural properties, and to some extent the possibility to apply it to models with parametric color class size, hence providing results that are valid for a family of similar models.

1.1 Definitions and Notation

SNs were introduced (with the name Well-Formed Nets) in [8]. It is a formalism, similar to Colored Petri Nets, featuring a syntax designed to naturally make symmetries explicit when the modelled system is symmetric (e.g. composed of several similarly behaving entities). A little SN model is depicted in Fig. 1, it is a small portion of a Distributed Memory fault tolerance mechanism model presented in [2]; the picture has been drawn with the GreatSPN GUI [1] and then (manually) translated into the textual format accepted by SNexpression (file with .sn extension). The automatic export from the GreatSPN GUI to the SNexpression net format is a planned future work. This is a natural choice since GreatSPN has been the first tool to support Symmetric Nets, moreover the GUI has been designed to allow extensions to the syntax (in SNexpression arc function terms may have both guards and filters) and handle several formalisms.


Results : RES
$\langle A_1, b, c, D_4\rangle$
$\langle a, b, c, D_4\rangle$
$\langle a\rangle$
Waiting : A
DM_PendRDReq : $TASK\_VAR$
$\langle a_{-1}\rangle$
$\langle a_{-1}, b\rangle$
$\langle a, b\rangle$
$[a \in A_1]$
$CheckCMP$
$\langle a, b\rangle$
$[d \in D_4]$
RD_MST
$\langle a_{-1}, b\rangle$
$\langle A_1 - a, a, b, c, D_4\rangle$
$\langle a_{-1}, a_{-2}, b, c, d\rangle$
DMmaster : $TASK\_VAR$
DMinput : $MBOX$

class $A = dmt\{1..3\}$ **is** $A_1 + ds\{1..2\}$ **is** $A_2 + sm\{1..3\}$ **is** A_3
class $D = \{\text{wr}\}$ **is** $D_1 + \{\text{rd}\}$ **is** $D_2 + \{\text{rdko}\}$ **is** $D_3 + \{\text{rdok}\}$ **is** D_4
class $B = dmv\{1..5\}$ **class** $C = val\{1..3\}$ **domain** $RES = A \times B \times C \times D$
domain $MBOX = A \times A \times B \times C \times D$ **domain** $TASK\_VAR = A \times B$

GreatSPN notation
$\langle A_1 - a, a, b, c, D_4\rangle$
$\langle A_1, b, c, D_4\rangle$
$[a \in A_1]$
$domain\ \ MBOX = A \times A \times B \times C \times D$
class$A = dmt\{1..3\}isA_1 + ds\{1..2\}isA_2$ $+sm\{1..3\}isA_3$
class$N$ circular $n1..4$
$[n_1{+}{+} \neq n_2]\langle c, n_1{+}{+}\rangle$
SNexpression CLI and .sn notation
$\langle S\_A\{1\} - a, a, b, c, S\_D\{4\}\rangle$
$\langle S\_A\{1\}, b, c, S\_D\{4\}\rangle$
$[a\ \ in\ \ A\{1\}]$
$MBOX := A\hat{}2, B, C, D$
$classA := \{3, 2, 3\}$
$setN\{4\}ordered$
$\langle c, !n\_1\rangle[!n\_1\ != n\_2]$

Fig. 1. A fragment of a distributed memory SSN model.

For the sake of space in this section we shall only describe in some detail the color structure of a SN, assuming that the reader is familiar with PN and HLPN formalisms and the definition of places, transitions, input, output and inhibitor arcs, marking. The color structure of a SN is built upon the basic color classes $\mathcal{C} = \{C_i, i = 1, \ldots, n\}$ which are finite and disjoint sets[2], may be (circularly) *ordered* or *partitioned into static subclasses* $C_{i,j}$. Transition and place *color domains* are defined as Cartesian products of classes : $D = \bigotimes C_i^{e_i}, e_i \geq 0, i = 1, \ldots, n$ (class C_i appears e_i times in the product). The color domain $cd(p)$ of place p defines the possible colors (tuples of color elements from $cd(p)$) of the tokens in its marking; the color domain $cd(t)$ of transition t defines its possible firing modes: these are

[2] In the tool color classes are denoted with a single capital letter: A, B, C, ... and the Cartesian product of classes is denoted as a comma separated list of classes.

tuples of color elements, and distinct typed variable names ($var(t)$) are used to refer to such elements in any tuple in $cd(t)$.

Let us consider the model in Fig. 1: $\mathcal{C} = \{A, B, C, D\}$, classes A and D are partitioned into static subclasses; the $cd(DMinput) = A^2, B, C, D$ and a tuple in this place could be $\langle dmt1, dmt2, dmv3, val2, rdok\rangle$; $cd(CheckCMP) = A, B, C$ ($var(CheckCMP) = \{a, b, c\}$) and one possible firing mode, also called instance, of this transition is $a = dmt1, b = dmv2, c = val1$. The enabling conditions of a transition instance and the effect of its firing depend on the functions on its input, inhibitor and output arcs. Guards can be associated with transitions, to constrain the set of valid instances. Transition $CheckCMP$ has several input and output places and its instances must satisfy predicate $a \in A_1$; the function on the arc from $DMinput$ is $\langle S\_A\{1\}-a, a, b, c, S\_D\{4\}\rangle$, while the functions on the arcs connecting it to place $DMmaster$ are $\langle a, b\rangle$. The domain of an arc function linking place p to transition t is $cd(t)$, whereas its codomain is $Bag[cd(p)]$[3]. Its general form is: $\sum_i \lambda_i.T_i[g_i]$, $\lambda_i \in \mathbb{N}$, where T_i is a function-tuple $\langle f_1, \ldots, f_k\rangle$ denoting the Cartesian product of *class functions* f_i. Each class-function f is a linear function defined on a subset of variables of $var(t)$ of the same type. Let $var_{C_i}(t)$ be the subset of $var(t)$ of type C_i, and $\widetilde{C_i}$ the set of static subclasses of C_i, then $f : cd(t) \rightarrow Bag[C_i]$ is so defined:

$$f = \sum_{v_k \in var_{C_i}(t)} \alpha_k.v_k + \sum_{v_k \in var_{C_i}(t)} \gamma_k.!v_k + \sum_{q \in \{1,\ldots,|\widetilde{C_i}|\}} \beta_q.S_{i,q} + \eta.S_i$$

where $\alpha_k, \gamma_k, \beta_q, \eta \in \mathbb{Z}$. $v_k \in var_{C_i}(t)$ in this context denotes the projection of a transition instance on the k^{th} element of type C_i in its color domain; symbol ! denotes the *successor* operator mapping the value of v_k to its successor (the type of v_k must be ordered). $S_{i,q}/S_i$ is a constant function mapping to the set $C_{i,q}/C_i$. Boolean expressions g_i (guards) on $var(t)$ may be associated with transitions or individual tuples; their terms are *standard predicates* checking whether two variables hold the same value, or if a variable "belongs" to a given static subclass; if g_i is false for a given transition instance, the associated tuple evaluates to the empty-multiset. Scalars in class-functions must be such that no negative coefficients result from the evaluation of any color satisfying the guard possibly associated with the corresponding tuple/transition. Figure 1 contains examples of arc expressions involving several classes; in the table examples of functions operating on ordered classes (see Fig. 4) are also shown.

The calculus on which SNexpression operates, handles expressions of a language ($\mathcal{L}$) introduced in [5], very similar to arc functions but with additional constraints and a couple of extensions: the constraints are on the basic class functions (only $v_i, !^n v_i, S - v_i, S_i, S_{i,q}$ are allowed) and on their coefficients which cannot be negative, while the extensions are the use of intersection of basic class functions as tuple elements, and the possibility to use a predicate also as a *filter* placed as a prefix in front of a tuple (filtering out the elements not satisfying

[3] A multiset on set D is a map $D \rightarrow \mathbb{N}$. $Bag[D]$ denotes the set of all multisets on D. If $m \in Bag[D]$, and $d \in D$, $m[d]$ is the multiplicity of d in m.

the filter predicate from the tuple evaluation). A number of unary and binary operators are defined on these expressions, which are useful when defining structural properties on SN models. The SNexpression tool implements a calculus on $\mathcal{L}$ and provides several off-the-shelf formulas to compute interesting structural properties of SN models.

To the best of our knowledge no other tools implement a general calculus for structural analysis of HLPNs. Even very advanced tools, e.g. Snoopy [9], take advantage of symmetry properties in the color structure to efficiently perform the net unfolding [10], but do not exploit it for structural analysis.

2 Tool Architecture and Functions

The architecture of SNexpression is organized in three layers, depicted in Fig. 2: the Library for Symbolic Calculus (LSC), the SN management framework (SNF), and the Command Line Interface (CLI). The LSC is a sort of Computer Algebra System that handles base-level SN expressions. The SNF middle layer manages more abstract objects, such as structural relation formulae, directly derived from a SN definition that may be loaded into the system; it also provides the algorithms needed to automatically derive the SODE for a given SSN model, based on a manipulation of SSN annotations.

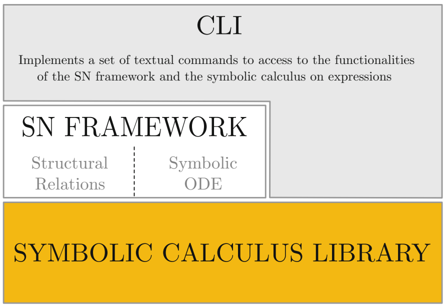

CLI
Implements a set of textual commands to access to the functionalities of the SN framework and the symbolic calculus on expressions
SN FRAMEWORK
Structural Relations
Symbolic ODE
SYMBOLIC CALCULUS LIBRARY

Fig. 2. Architecture of SNexpression.

The CLI is a shell surrounding both the library and the SN framework, through which the user can operate directly on base-level expressions, using the CLI as a sort of symbolic calculator, or at a higher level, performing structural analysis of (S)SN models previously loaded.

To help the users operation during a session the CLI provides a multipage textual notebook where it is possible to annotate and save formulae to be submitted, or results, or comments, in other words anything useful to support multi-step complex analysis. Since the format of the LSC output is pretty similar to the syntax of CLI input, copy-and-paste from the notebook to the command window and vice-versa may be conveniently used.

SNexpression is implemented in Java. The LSC is distributed as a standalone `jar` file, so that programmers can use it in other projects. Its API is available at URL: www.di.unito.it/~depierro/SNexpression/libAPI, we plan to make the LSC soon available as open-source project. At the current release, the CLI and the SNF are built as a unique executable, but we plan to make also the SNF accessible through a public API. Since the CLI reads from the standard input, it might be integrated in other tools. The following sections discuss the functions of the three layers of the tool architecture in more detail.

2.1 The LSC: A Computer Algebra System for SN Expressions

The major functionalities and the design of the LSC are summarized here. The library implements a (parametric) rewriting system: algebraic rules are used to rewrite symbolic expressions composed of terms of $\mathcal{L}$, and the associated set of operators: sum, difference, intersection, composition, transpose. Rewriting stops when no more rules apply, in which case the resulting term is considered in "normal form". Final expressions manipulated by the LSC match a sort of *disjunctive normal form*, where only SN functions, guards, filters and sum/intersection operators may occur.

With respect to earlier versions, the LSC currently supports both set- and bag-expressions (i.e. espressions returning multisets), with the only exception of composition, which is partially implemented for bag-expressions: its complete definition is work in progress. The support operator provides a convenient bridge between bag- and set-expressions.

Thanks to its modular layout and intuitive API, the LSC may also be used as a standalone component. As a direct consequence of its design, it is possible to directly build and manipulate objects (terms) at three different levels: class-functions, guards/filters, and function-tuples. Each level has its own set of operators, similar among the levels. Guards/Filters (standard predicates), class-functions, and single function-tuples have a canonical representative, which coincides with their normal form. There is no canonical form for sums (bags) of function-tuples: in general, however, equivalence between expressions may be syntactically checked by using the difference operator.

Library Architecture and API. The library consists of around one hundred `Java` classes/interfaces, divided in ten packages. The adopted design has many advantages. *Ease of extension/maintenance:* changes or updates (e.g., adding new language elements) are low-cost. *Modular testing/debugging:* every single element of the language can be managed in a uniform way. *Efficiency:* term normalization complexity is alleviated by a reduced use of recursion (the normalization times for many examples, some of which very complex, vary from msec. to sec.).

A code snippet illustrating the several steps needed to create and normalize a SN expression (the transpose of a tuple composition) is listed in Fig. 3. A (simplified) UML class-diagram describing the top of the LSC type hierarchy, and its connections to the lower levels, can be found in the tool home page, together with a small portion of the library's API concerning simplification methods.

2.2 The Symmetric Nets Management Framework

The SNF implements the method to load a SN description and those to compute some symbolic structural relations on it, listed in Table 1: some relations are functions on sets, others return multisets. For the structural relations computation it exploits the functions implemented in the library (difference, transpose, composition) on one hand, and the information on the loaded SN structure on the other hand: the model structure allows to select transition pairs that might be in

```
import wncalculus.color.ColorClass;
import wncalculus.classfunction.*;
import wncalculus.guard.*;
import wncalculus.tuple.*;
//...
ColorClass C = new ColorClass("C", true); //ordered class C s.t. |C| > 1
Interval i1 = new Interval(3,8), // [3,8] (constraint)
         i2 = new Interval(2,2); // [2,2]
ColorClass D = new ColorClass("D", new Interval[] {i1, i2}); // partitioned
    class D = D{1} \cup D{2}
Projection c_1= Projection.builder(1, C), // c_1
           c_2= Projection.builder(2, 1, C), // !c_2
           d_1= Projection.builder(1, D); // d_1
SetFunction comp_c_1, comp_d_1, sd2, inter;
comp_c_1 = ProjectionComp.factory(c_1); // S - c_1
comp_d_1 = ProjectionComp.factory(d_1); // S - d_1
sd2 = Subcl.factory(2, D); // constant D{2}
inter = Intersection.factory(comp_d_1,sd2);//D{2} \cap S -d1
Domain dom = new Domain(C,C,D); // domain C^2 x D
Guard g1 = Membership.build(d_1,sd2,true,dom); //d_1 \in D{2}
Tuple t1 , t2;
t1 = new Tuple(dom, c_1, comp_c_1, c_2, d_1); // <c_1,S-c_1,!c_2,d_1>
t2 = new Tuple(null, g1, dom, comp_c_1, c_2 , inter); // <S-c_1,!c_2,(S-d_1
    * S_D{2})>[d_1 in D{2}]
TupleComposition tcom = new TupleComposition(t1, t2);
TupleTranspose tcom_trans = new TupleTranspose(tcom);
List<LogicalExpr> result = tcom_trans.simplify();
System.out.println(result);
```

Fig. 3. Snippet showing creation and simplification of "function-tuple" expressions.

structural conflict or causal connection relation, then the arc functions labeling the involved arcs are processed through the symbolic calculus implemented by the LSC. For the mutual exclusion structural relation an ad-hoc computation algorithm [7] is applied after pre-processing the selected arc expressions through the library methods.

Finally, the SNF implements the algorithm to derive a set of Symbolic Ordinary Differential Equations from a (partially unfolded) SN [3,4]: it exploits the library to compute the (multiset) transpose of the arc expressions and to derive the *enabling degree* of homogeneous sets of transition instances. It operates with just one command *print_ode* after having loaded the SN to be translated. A file .ode is produced, including the set of SODE (one for each model place) ready to be solved using Rstudio.

2.3 The Command Line Interface

The CLI is the user interface of the tool: despite its simplicity it provides the essential commands to access the main functions of the LSC and of the SNF implementing four kinds of commands: definition of classes or language expressions; application of operators to expressions and simplification; loading a SN and computing some structural relations on it; derivation of a set of SODE from a SN, which in turn needs the computation of some structural relations. The syntax of all commands is described in the manual: Table 1 summarizes the main types of commands; a few detailed examples are described in Sect. 3. By convention the *color classes* are denoted with capital letters $(A, B, C, ...)$ while small letters, possibly indexed with an integer, denote *variables* whose type is the corresponding capital letter class (e.g. a or a_2 of type A). Classes may be partitioned into static subclasses denoted by the class capital letter followed by an integer index (e.g. $A\{1\}$ subclass of A). Classes have finite cardinality, but it can be defined to be parametric (by default any class has a parametric cardinality n greater than or equal to two; when a class is partitioned into static subclasses only one subclass may have parametric cardinality). Domains are Cartesian products of color classes, if one class appears more than once in a domain, it is listed once followed by the number of repetitions.

Table 1. Summary of the main commands implemented in the CLI.

Description	CLI syntax (examples)
Defining a class and showing its definition	
A (possibly ordered) class	$set\ C\ ordered$
and its subclasses	$set\ M := \{10, 3, [4, n]\}$
Show class definition	$class(A)$
Symbol definition	
Define domain symbol	$dom := A\hat{}2, B, C\hat{}3$
Define expression symbol	$exp := @A\hat{}3 < a\_1 * S - a\_2, (a\_1 + a\_2) * S - a\_3 >$
(with domain prefix)	($*$: intersection, a_i and $S - a\_j$: basic functions)
may include a *filter*	$@D\hat{}2[d\_1! = d\_2, d\_2\ in\ D\{1\}] < d\_2, S - d\_1 >$
and a *guard*	$@A\hat{}3 < a\_1, a\_3 > [a\_3\ in\ M\{2\}]$
Define multiset expression	$mexp := @C, D\ mset : 2 < d\_1, c\_1 > + < S\_D, c\_1 >$
Operators application	
support (applies to bag-expressions)	$<<mset\ expression>>$
transpose	$expression'$
difference	$expression1 - expression2$
composition	$expression1.expression2$
	implementation for bag-expressions is not yet complete
simplify expression	$s(e)$ rewrites an expression into a normalized form
simplify and fold	$sf(e)$ merges terms or expressions ($\neq$ constraints)
Symmetric Nets Framework commands	
Nets management commands	
Load net	load "DistMem.sn"
Input/output/inhib. places and arc expr	I(t) / O(t) / H(t)
Symbols for arc (bag-)expressions	I(t,p) / O(t,p) / H(t,p)
Structural properties	
Conflict	$SC(t1, t2, p)$ or $SC(t1, t2)$
Self-Conflict	$SC(t, p)$ or $SC(t)$
Causal Connection	$SCC(t1, t2, p)$
Mutual Exclusion	$SME(t1, t2, p)$
Added By (set/ multiset)	$AB/ABmset(t, p)$
Removed By (set/multiset)	$RB/RBmset(t, p)$
Derivation of a set of SODE	$print\_ode$

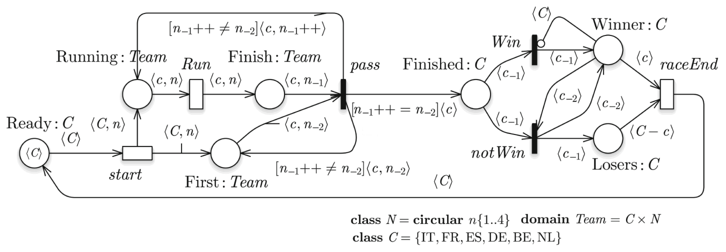
The expressions can be interpreted as functions mapping into multisets or functions mapping on sets, the latter case is the default. The prefix *mset:* indicates that an expression denotes a function mapping on multisets. The expressions syntax takes the form of a *sum of tuples*, each tuple may be prefixed with a *filter* and suffixed with a *guard* (both filter and guard take the form of SN *standard predicates*). The tuple elements are intersections ($*$) of basic functions: projection, complement, successor, constant (whole class or one static subclass).

Operators can be applied to expressions: there are two unary operators, support and transpose, and two binary operators, difference and composition. The support operator can be applied to a multiset-expression to obtain the corresponding expression mapping on sets; the transpose operator is available both for expressions mapping on multisets and for those mapping on sets: the result of its application is an expression of the same type. The difference can be applied to any pair of expressions (of the same type) while the composition is completely implemented for expressions mapping on sets and on a significant subset of multiset expressions. These operators are the basis for the SN structural analysis implemented in the SN Management Framework. For instance the structural conflict between two transitions t_1 and t_2 sharing an input place p is computed as follows: $SC(t_1, t_2, p) = <<I(t_1, p) - O(t_1, p)>>'.<<I(t_2, p)>> + << O(t_1, p) - I(t_1, p)>>'.<< H(t_2, p)>>$ where $I(t, p)$ and $O(t, p)$ are respectively the expressions on the input and output arcs connecting p and t. This could be useful to identify the groups of immediate transitions that are potentially in conflict and define how such conflicts should be solved.

To support the user in performing experiments with the tool, the CLI embeds a multi-page notebook: it is possible to copy-and-paste commands annotated in the notebook to the command window and then copy-and-paste results from the command window back in the notebook. When the color classes involved in the expressions processed by the tool have parametric cardinality, the result of a computation is not a single expression but a list of expressions, each with associated a different range of possible values for the classes cardinality: indeed one of the strong points of the tool is its ability to handle expressions without necessarily fixing the color classes sizes, so that the obtained result is valid for a family of models differing only in the size of (some) color classes.

3 Use Cases: Exploiting SNexpression

The goal of this section is to show on a few practical examples some features of SNexpression. Let us consider the relay race model in Fig. 4 (described in [7]), representing a set of teams (class C), each composed of four athletes (ordered class N, $|N| = 4$), competing in a relay race. Some symbolic structural properties of interest are the *causal connection* and *structural conflict* involving transition *Run* and the immediate transitions *pass*, Win and $notWin$; these properties



$[n_{-1}{+}{+} \neq n_{-2}]\langle c, n_{-1}{+}{+}\rangle$
Running : Team
Run
Finish : Team
pass
Finished : C
Win
Winner : C
raceEnd
notWin
Losers : C
Ready : C
start
First : Team
$\langle c, n\rangle$
$\langle c, n_{-1}\rangle$
$\langle c, n_{-2}\rangle$
$\langle C, n\rangle$
$\langle C\rangle$
$\langle c_{-1}\rangle$
$\langle c_{-2}\rangle$
$\langle c\rangle$
$\langle C - c\rangle$
$[n_{-1}{+}{+} = n_{-2}]\langle c\rangle$
$[n_{-1}{+}{+} \neq n_{-2}]\langle c, n_{-2}\rangle$
class N = circular $n\{1..4\}$ domain $Team = C \times N$
class $C = \{$IT, FR, ES, DE, BE, NL$\}$

Fig. 4. An SN model of a relay race.

are useful (among others) for model validation purposes, or to correctly define immediate transition weights. Let us consider the commands summary in Table 2

Table 2. Examples of structural relation expressions.

1) $SCC(pass, Run, Running) = @C, N\langle c\_1, !3n\_1, S - n\_1\rangle$
2) $SCC(pass, Win, Finished) = @C[n\_1 = !3n\_2]\langle c\_1, S\_N, S\_N\rangle$
3) $SC(Win) = @C\langle S - c\_1\rangle$
4) $SME(Win, notWin, Winner) = @C, N\,\hat{}\,2\langle S\_C\rangle$
5) $s(fscc2.fsc3) = @C[n\_1 = !3n\_2]\langle S - c\_1, S\_N, S\_N\rangle$
where $fscc2 := @C[n\_1 = !3n\_2]\langle c\_1, S\_N, S\_N\rangle$, $fsc3 := @C\langle S - c\rangle$
6) $SC(RD\_MST, CheckCMP) = (\langle a\_1, S\_A, S\_B, S\_C, S\_D\{4\}\rangle$ $+\langle (S\_A\{1\} * S - a\_1), a\_1, b\_1, c\_1, S\_D\{4\}\rangle)[a\_1\ in\ A\{1\}]$
7) $SC(CheckCMP, RD\_MST) = \langle a\_2, b\_1, c\_1\rangle[a\_1! = a\_2, d\_1\ in\ D\{4\},$ $a\_1\ in\ A\{1\}, a\_2\ in\ A\{1\}] + \langle a\_1, S\_B, S\_C\rangle[d\_1\ in\ D\{4\}, a\_1\ in\ A\{1\}]$

1) computes the instances $\langle c, n', n''\rangle$ of *pass* that may enable an instance $\langle c\_1, n\_1\rangle$ of *Run*: through *Running*; the result shows that such instances involve an athlete of the same team (c_1) with sequence number n' equal to the predecessor[4] of n_1, provided that $n'' \neq n\_1$ (i.e. the team has not run the last section yet). 2) computes the instances $\langle c, n', n''\rangle$ of *pass* enabling instance $\langle c\_1\rangle$ of Win: the result has a filter and denotes the instances involving the same team c_1, and the last section runner (the predecessor of n''). 3) computes the structural auto-conflicts among different instances of Win, while the result of 4) shows that Win and $notWin$ are mutually exclusive: indeed, $Winner$ is input place for $notWin$ and inhibitor place (with arc function $\langle S_C\rangle$) for Win. In SSNs with immediate transitions it is useful to check for confusion, i.e., situations where the model is underspecified (a situation solved by using priorities). In our example, this may arise due to the fact that different instances of Win are in conflict with each other. There would be confusion if an instance of *pass* fired while an instance of Win is enabled: this could cause the enabling of another instance of Win in conflict with the former. 5) shows how to obtain the *confusing* instances of *pass* by composing the results of 2) and 3): in this case the SNF is not involved.

Other structural relations can be computed on the model in Fig. 1, whose arc functions are a bit more complex as illustrated by relations 6) and 7) in Table 2. The resulting expressions have the same domain as the 2nd parameter of SC, namely A, B, C for 6) and $A\,\hat{}\,2, B, C, D$ for 7). The terms are pair-wise disjoint: this enhances readability and interpretation.

[4] Since $|N| = 4$ the predecessor $!^{-1}n$ of n coincides with the third successor $!^3n$.

The tool can be used for other purposes. A recent implementation concerns the automatic generation of Symbolic ODE from an SSN model (command *print_ode*). The technique applies only to models whose underlying stochastic process is density dependent. One condition for an SSN to satisfy such property is the coverage of places by P-invariants. In [4] an application to a botnet model has been illustrated (this is one of the examples that can be downloaded from the tool's web page). SNexpression can be used to check if a given P-indexed vector of multiset expressions defines a set of *colored* P-invariants, and possibly prove the coverage of all place instances. The expressions in the P-indexed vector denote functions from the place color domains to the P-invariant's domain. An example of P-vector of expressions $C, L \rightarrow Bag[L]$ is: $pinv[NoConBot] := @C, L\ mset : \langle l \rangle [c\ in\ C\{1\} + c\ in\ C\{2\}]$; $pinv[ConBot] := @C, L\ mset : \langle l \rangle [c\ in\ C\{2\}]$; $pinv[InactBot] := @C, L\ mset : \langle l \rangle [c\ in\ C\{3\} + c\ in\ C\{4\}]$; $pinv[ActBot] := @C, L\ mset : \langle l \rangle [c\ in\ C\{3\} + c\ in\ C\{4\}]$. In order to prove that this vector corresponds to a set of P-invariants we need to show, for each transition t, that the sum over all places of the compositions of P-invariant's function ($pinv[p]$) with the difference $O(t,p) - I(t,p)$ results in the null function. Due to space constraints we show only the result for transition *ConnectBot*: $s(pinv[NoConBot].f4 + pinv[ConBot].f5) = null$ where $f4 := @C, Lmset :< 0_C, 0_L > -I(ConnectBotNet, NoConnBot)$ and $f5 := O(ConnectBotNet, ConnBot)$. The same result holds for all transitions, thus $pinv[]$ is a P-invariant: it represents $|L|$ invariants, indicating that the number of tokens with second component $l \in L$ is constant, and have the correct label in C. In the tool web page other P-invariants for this model are available.

4 Conclusions and Installation Instructions

SNexpression implements a symbolic calculus useful for studying the structure of a Symmetric Net and deriving information on its behavioral properties. It has also been used to derive a set of SODE from an SSN model for performance analysis purposes. Several extensions are planned: completing the composition of bag-expressions, further automation of net structural calculus (e.g., checking P-invariants or building Extended Conflict Sets), automatizing the SN partial unfolding procedure which is a preliminary step for the generation of the SODE from a SN model, providing access to the different software layers with suitable APIs. Finally, we plan to build a bridge between GreatSPN and SNexpression.

A free version of the tool is available: download the archive `SNEx.zip` from the project homepage www.di.unito.it/~depierro/SNexpression, unzip its content into a folder. The extracted file structure contains the main program `SnexCLI.jar` and, in the folder `lib`, the library `SNexLib.jar`. To launch the tool in an OS shell run: `java -jar <path to SNexCLI.jar>` (JRE 1.8 or above is necessary). At the project's web page, the user can find a reference manual and some examples to immediately start using it.

References

1. Amparore, E.G., Balbo, G., Beccuti, M., Donatelli, S., Franceschinis, G.: 30 years of GreatSPN. In: Fiondella, L., Puliafito, A. (eds.) Principles of Performance and Reliability Modeling and Evaluation, pp. 227–254. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30599-8_9
2. Ballarini, P., Capra, L., Franceschinis, G., De Pierro, M.: Memory fault tolerance software mechanisms: design and configuration support through SWN models. In: 3rd International Conference on Application of Concurrency to System Design ACSD 2003, Guimaraes, Portugal, 18–20 June 2003, pp. 111–121 (2003)
3. Beccuti, M., Capra, L., De Pierro, M., Franceschinis, G., Follia, L., Pernice, S.: A tool for the automatic derivation of symbolic ODE from symmetric net models. In: 27th IEEE International Symposium MASCOTS 2019, Rennes, France, October 21–25 2019, pp. 36–48 (2019)
4. Beccuti, M., Capra, L., De Pierro, M., Franceschinis, G., Pernice, S.: Deriving symbolic ordinary differential equations from stochastic symmetric nets without unfolding. In: Bakhshi, R., Ballarini, P., Barbot, B., Castel-Taleb, H., Remke, A. (eds.) EPEW 2018. LNCS, vol. 11178, pp. 30–45. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02227-3_3
5. Capra, L., De Pierro, M., Franceschinis, G.: A high level language for structural relations in well-formed nets. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. a high level language for structural relations in well-formed nets, vol. 3536, pp. 168–187. Springer, Heidelberg (2005). https://doi.org/10.1007/11494744_11
6. Capra, L., De Pierro, M., Franceschinis, G.: A tool for symbolic manipulation of arc functions in symmetric net models. In: Proceedings of the 7th International Conference VALUETOOLS 2013, Torino, Italy, pp. 320–323, ICST. Belgium (2013)
7. Capra, L., De Pierro, M., Franceschinis, G.: Computing structural properties of symmetric nets. In: Proceedings of the 15th International Conference on Quantitative Evaluation of Systems, QEST 15, Madrid, ES. IEEE CS (2015)
8. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: Stochastic well-formed coloured nets for symmetric modelling applications. IEEE Trans. Comput. **42**(11), 1343–1360 (1993)
9. Heiner, M., Herajy, M., Liu, F., Rohr, C., Schwarick, M.: Snoopy – a unifying petri net tool. In: Haddad, S., Pomello, L. (eds.) PETRI NETS 2012. LNCS, vol. 7347, pp. 398–407. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31131-4_22
10. Liu, F., Heiner, M., Yang, M.: An efficient method for unfolding colored Petri nets. In: Winter Simulation Conference, WSC 2012, Berlin, Germany, 9–12 December 2012, pp. 295:1–295:12 (2012)