Model Analysis via a Translation Schema to Coloured Petri Nets

Visar Januzaj[1] and Stefan Kugele[2][1]

[1] Technische Universität Darmstadt, Fachbereich Informatik,
FG Formal Methods in Systems Engineering,
Hochschulstr. 10, 64289 Darmstadt, Germany
`{januzaj, kugele}@forsyte.de`

[2] Technische Universität München, Institut für Informatik,
Boltzmannstr. 3, 85748 Garching bei München, Germany
`kugele@in.tum.de`

Abstract. Model-driven development (MDD) has become a success story and a de facto standard in the development of safety-critical embedded systems. The daily work in the development of such systems cannot be imagined without industry standard CASE tools like e.g. MATLAB/Simulink. Often however, the analysis capabilities of such tools are limited. Therefore, we propose to combine them with the powerful analysis tools developed for Coloured Petri Nets (CPNs).
In this paper, we present a translation schema from COLA—a synchronous data-flow language—to CPNs. We believe this approach to be also feasible for other data-flow languages as long as they have a well-defined syntax and semantics. The combination of both modelling languages allows us to verify properties of COLA models using algorithms and tools designed for CPNs. An example demonstrates the viability of this approach.

Key words: Coloured Petri Nets, Model-driven development (MDD), embedded systems, synchronous data-flow languages

1 Introduction

Embedded systems development is seeing a surge of interest both in academia and industry, this is due to the rapid growth of the market share of embedded systems. About 98% [1] of all processors are nowadays used in embedded systems. Their presence becomes ubiquitous, ranging from portable music players and mobile phones to airbag controllers and flight control systems (FCS). For the first mentioned consumer electronics products properties like reliability, robustness, and correctness are circumstantial. Whereas in the field of automotive or avionics control software, failures of any kind may be fatal or at least result in large warranty costs.

A multitude of different development tools, frequently based on model-driven development (MDD) approaches, have been invented to tackle the complexity of embedded systems design. Nowadays, modelling large system designs without industry standard CASE tools like e.g. MATLAB/Simulink [2] or SCADE by Esterel Technologies (A380, FCS) [3] cannot be imagined. Here, most different aspects play a major role: due to abstraction and different model views, the complexity apparent to the system's developer is reduced. This helps to reduce design errors by supporting the engineer at the daily work. In the desirable case, that the chosen modelling technique (language) has a well-defined basis, the use of formal methods is facilitated. This improves the quality of the system de novo. In the automotive domain for instance, OEMs are working in a highly competitive mass market, where the time to market is essential for the success of a product and for the company as a last consequence. There, a reduction or—in the best case—the absence of errors detected late in the overall development process considerably saves money and shortens the development process, which in return reduce development costs.

Since the widely used MATLAB/Simulink lacks a formally-defined semantics [4], in a co-operation project together with an industry partner from the automotive domain, the synchronous data-flow language COLA (Component Language) [5] has been developed. During its development, aims like usability, soundness, and reusability were in mind. Around this language, a fully integrated tool [6–14] was created supporting the complete development process ranging from the early requirements engineering, the system modelling, to the system deployment phase.

Coloured Petri Nets (CPNs) [15–17]—similar to COLA—are a graphical modelling language emerged from the combination of Petri Nets [18] and the functional programming language Standard ML (SML) [19,20]. CPNs and their corresponding computer tools (*CPN Tools* [21]) have been successfully applied in various application areas and industry projects [17], ranging from VLSI chip design, communication protocols [22] to military systems [23–25].

In this paper, we describe how to benefit from both modelling techniques by first translating COLA designs into the CPN formalism and second using analysis techniques and tools applicable to Coloured Petri Nets. This combination allows to augment the well-defined COLA syntax and semantics with the comprehensive CPN Tools.

1.1 Related work

Coloured Petri Nets have been extensively used to model and verify business processes. Gottschalk et al. [26] translated Protos models, i. e. a popular tool for business process modelling, into Coloured Petri Nets for simulation, testing, and configuration reasons. Moreover, in the field of Web services, CPNs are used. There, questions concerning correctness and reliability arise, when composing single Web services to more complex ones. Kang et at. [27] and Yang et al. [28] have studied the translation of WS-BPEL (*Web Services Business Process Execution Language*) or BPEL specifications into CPNs. This allows for analysis and verification of the composed Web services using for example CPN Tools [21]. However their translations are rather informal than a formal defined translation schema. Hinz et al. [29] translated BPEL specifications into Petri Nets in order to use the model checking tool LoLA to verify relevant properties.

Akin to the presented approach, the authors of [30] bring together the two modelling languages UML and CPN. They translate Use Cases and UML 2.0 Sequence Diagrams into CPN models for formal analysis. In the automotive domain or in the field of embedded systems design in general, *Live Sequence Charts (LSC)* are widely used as specification language. The authors of [31] claim, that LSC do not provide the possibility for analysis and verification and thus a translation into CPN is appropriate and which is given in a well-defined formal way.

1.2 Organisation

The remainder of this work is structured as follows: First, we will give a brief introduction into Coloured Petri Nets in Sect. 2 followed by a more detailed description of the Component Language COLA in Sect. 3. Sect. 4 presents the basic contribution of this work, namely a translation schema from COLA to CPN. An Example demonstrating the feasibility of this approach is given in Sect. 5. Finally, we conclude in Sect. 6.

2 Coloured Petri Nets

Coloured Petri Nets (CPNs) [15–17] belong to the family of high-level Petri nets. Their modelling power strongly relies on the composition between Petri nets and the high-level functional programming language Standard ML (SML) [19,20]. On the one hand, the usage of Petri nets offers an effective framework for *modelling* concurrency, communication, and synchronisation. On the other hand, the application of SML facilitates the *definition* and *manipulation* of the data. In order to be able to cope with the normally large size of real life systems and to introduce a better system overview, CPNs offer the possibility of hierarchically modelling, i. e. parts of the model are combined into submodules. In addition to the possibility of modular modelling, CPNs include a time concept. The integration of the notion of time allows the investigation of especially for distributed, real-time and embedded systems important timing requirements, such as deadline and delay constraints. Furthermore, a set of graphical computer tools is developed to support the modelling, editing, simulation, and analysis of various important properties of systems modelled as CPNs. This set of tools is integrated into the *CPN Tools* [21] framework.

In the following we briefly introduce the definitions of hierarchical and non-hierarchical CPNs. These definitions should serve to easier understand the terminology used for the translation of COLA models. For more detailed and complete definitions see [15–17]. We assume, however, that the reader is familiar with Petri nets and CPNs as well as with the notions such as *multi-sets* (multiple appearances of the same element, denoted *MS*), *marking* (a function mapping each place into a multi-set of tokens of the same colour), *reachability*, *firing*, etc.

Definition 1 (Non-hierarchical CPN (cf. [16])). *A non-hierarchical CPN is a 9-tuple $\Omega = (P, T, A, \Sigma, V, C, G, E, I)$ with:*

- *A finite set of* ***places*** *P and* ***transitions*** *T such that $P \cap T = \varnothing$.*
- *A set of directed* ***arcs*** *$A \subseteq P \times T \cup T \times P$.*
- *A finite set of* ***colour sets*** *Σ.*
- *A finite set of* ***variables*** *V, $Type(v) \in \Sigma, \forall\, v \in V$.*
- *A* ***colour set function*** *$C : P \to \Sigma$, $C(p) \in \Sigma, \forall\, p \in P$.*
- *A* ***guard function*** *$G : T \to Expr$, $Type(G(t)) = \texttt{BOOL}, \forall\, t \in T$.*
- *An* ***arc expression function*** *$E : A \to Expr$,
  $Type(E(a)) = C(p)_{MS}, \forall\, a \in A$ and a is connected to $p \in P$.*
- *An* ***initialisation function*** *$I : P \to Expr$, $Type(I(p)) = C(p)_{MS}, \forall\, p \in P$.*

In order to easier understand the following definition, we need to introduce some basic notions: *substitution transitions* are transitions that represent an abstraction of a more detailed submodule of a CPN system. The set of places belonging to the preset and the postset of a transition t is denoted $X(t) = {}^{\bullet}t \cup t^{\bullet}$. *Socket nodes* are called the places p surrounding a substitution transition t, i. e. $p \in X(t)$. The *socket type* function ST is defined as follows (cf. [16]):

$$ST(p,t) = \begin{cases} in & \text{if } p \in ({}^{\bullet}t \text{ - } t^{\bullet}) \\ out & \text{if } p \in (t^{\bullet} - {}^{\bullet}t) \\ i/o & \text{if } p \in ({}^{\bullet}t \cap t^{\bullet}) \end{cases}$$

Definition 2 (Hierarchical CPN (cf. [16])). *A hierarchical CPN is a 9-tuple $\Omega_H = (S, SN, SA, PN, PT, PA, FS, FT, PP)$ with:*

- *A finite set of* ***pages*** *S. Each page is a non-hierarchical CPN:
  $\forall\, s \in S, s = (P_s, T_s, A_s, \Sigma_s, V_s, C_s, G_s, E_s, I_s)$, the set of net elements of each page pair are disjoint.*
- *A set of* ***substitution nodes*** *$SN \subseteq T$ - the set of all transitions in Ω_H.*
- *A* ***page assignment function*** *$SA : SN \to S$, such that no page is a subpage of itself.*
- *A set of* ***port nodes*** *$PN \subseteq P$ - the set of all places in the entire Ω_H.*
- *A* ***port type function*** *$PT : PN \to \{in, out, i/o, general\}$.*
- *A* ***port assignment function*** *$PA : SN \to Pot(X(SN) \times PN)$ such that:*
  - *The relation between socket nodes and port nodes is defined as follow:
    $\forall\, t \in SN$: $PA(t) \subseteq X(t) \times PN_{SA(t)}$.*
  - *Correct types for socket nodes are required:
    $\forall\, t \in SN, \forall (p_1, p_2) \in PA(t) : [PT(p_2) \neq general \Rightarrow ST(p_1, t) = PT(p_2)]$.*
  - *Related nodes have the same colour set and initialisation:
    $\forall\, t \in SN, \forall (p_1, p_2) \in PA(t) : [C(p_1) = C(p_2) \wedge I(p_1) <>= I(p_2) <>]$.*
- *A finite set of* ***fusion sets*** *$FS \subseteq P_s$, such that all elements have the same colour set and initialisation.*
- *A* ***fusion type function*** *$FT : FS \to \{global, page, instance\}$, such that page and instance fusion sets belong to a single page.*
- *A multi-set of* ***prime pages*** *$PP \in S_{MS}$.*

3 COLA—The Component Language

The key concept of COLA is that of *units*. Consequently, all COLA models are built up by simple units—so-called *basic blocks*. They are at the lowest level in a hierarchy of composed units. Those composed units are called *networks* and are used to build up more complex data-flow networks. The basic blocks are atomic, i. e. they cannot be further decomposed. They define the basic (arithmetic and boolean) functions of a system. Environmental interaction is given via typed *ports*. Port to port communication is established via *channels*. According to the synchronous paradigm, which COLA is based on, computation and communication takes no time. Thus, cycles realised by channels that connect an output port to an input port of the same network have to contain at least a *delay* block. It has an initial value and defers value propagation by one *clock tick*. This construct is well-suited to realise memory and feedback loops often used in control systems.

In addition to basic blocks and networks, units can be decomposed into *automata*, i. e. finite state machines similar to Statecharts [32]. The behaviour in each state is again determined by units corresponding to each of the states. This capability is well-suited to express disjoint system modes, also called *operating modes* (cf. [33, 34]).

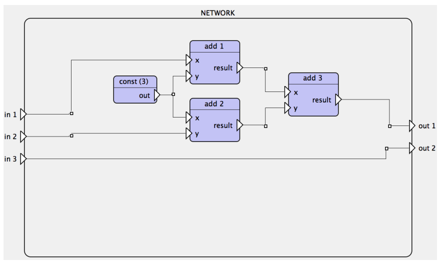
Figure 1(a) shows a COLA network consisting of a couple of add blocks and a const block with value 3. An impression of a similarly behaving CPN is given in Fig. 1(c). The top level of the latter is depicted in Fig. 1(b). In the following, we give a formal definition of each COLA language construct in order to develop a formal translation schema into Coloured Petri Nets.

3.1 COLA in Detail

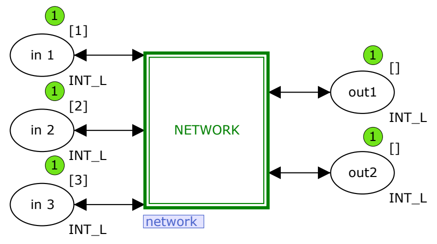
COLA models are build up by very few, but powerful primitives. The definition of COLA language elements is rather short and builds upon other industry standard data-flow language elements found in MATLAB/Simulink, SCADE, or Lucid Synchrone. COLA's advantage, however, is the reduction to only the bare necessities. Moreover, this slender syntactical core is well-defined and provides a rigourous semantics. The authors of COLA [5] defined the semantics by providing an interpreter for COLA that can be seen as a reference implementation. Moreover, a graphical as well as a textual syntax definition is given in the mentioned article. This formal framework is required, e. g. to simulate the model in a well-defined way, or to perform static analysis like type checking and behavioural verification.

The following sections provide a definition of each syntactical language element.

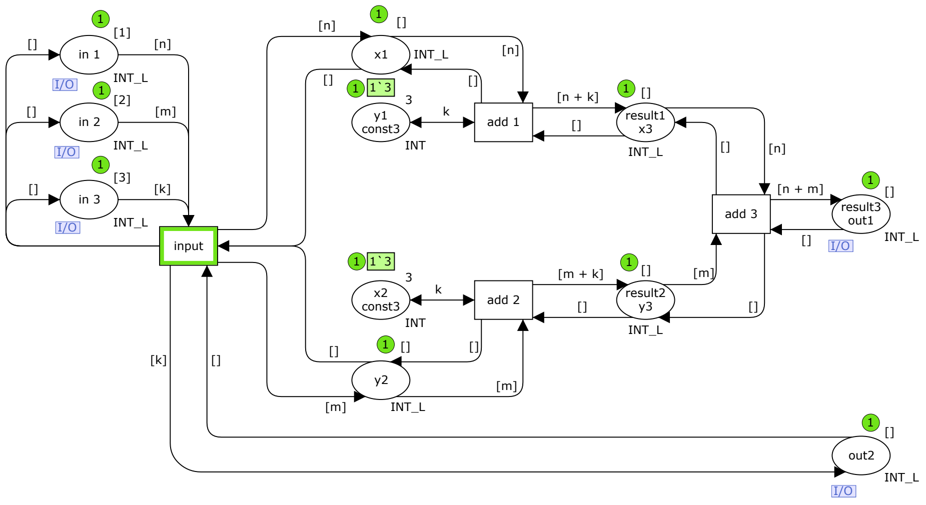
Units define a relation between its typed input and output *ports*. Ports are used for environment interaction. The combination of all input ports $P_{in} =$



(a) COLA network



(b) CPN model (superpage *CPNnetwork*)



(c) CPN model (subpage *network*)

Fig. 1. (a) A COLA network consisting of a couple of basic blocks. (b) The top level of the corresponding CPN with the same behaviour (c).

$\langle i_1 : t_1, \ldots, i_m : t_m\rangle$ where t_j, $1 \leq j \leq m$, is the type of port i_j, and all output ports $P_{out} = \langle o_1 : t_{m+1}, \ldots, o_n : t_{m+n}\rangle$, with $m, n \in \mathbb{N}$, defines the unit's *signature* $\sigma = (P_{in} \rightarrowtail P_{out})$.

A unit u is defined as a 3-tuple $\langle n, \sigma, I\rangle$ where n is its name, σ defines the signature, and I specifies the actual implementation. A unit can be considered as the superclass for special types of units, namely *functional block*, *timing block*, *network*, and *automaton*. Depending on the used type, the implementation I is chosen adequately. A detailed description of the different unit types is presented next.

Functional blocks can realise a multitude of different operators: first, fundamental arithmetic operations ($+$, $-$, $/$, $*$) can be used. Second, COLA provides the basic comparison operators ($=$, $\neq$, $<$, $\leq$, $>$, and $\geq$). Third, Boolean connectivities are supported ($\wedge$, $\vee$, and $\neg$). A functional block u is defined as follows:

$u = \langle n, (\langle lop : t, rop : t \rangle \rightarrowtail \langle result : b \rangle), I \rangle$. All operations are binary with input ports *lop* (*left operand*) and *rop* (*right operand*) and provide a result port *result*. The type b of the result depends on the operation. For arithmetic operations holds that b is equal to t, e. g. `Int` or `Real`. All other operators return a result value of type `Boolean`. Their implementation I is defined by the used operator, i. e. the functional block `add`, cf. Fig. 2(a), (operator $+$) for example is mathematically defined and implemented as $result := lop + rop$.

Delays (timing blocks) retaining a value for a single time unit (tick) and thereby provide a low-level realisation of variables as found in high-level programming languages. This is indispensable in the context of feedback control system. There, computed values have to be stored and fed-back as input for the next clock tick. Initially, delays are initialised with a constant as default value. Each cycle in a network has to contain at least one delay. Otherwise, the modelled system cannot be interpreted.

A delay is a unit, defined in the following way: $d = \langle n, \sigma, I \rangle$, with n being an identifier, the signature $\sigma = (\langle in : t \rangle \rightarrowtail \langle out : t \rangle)$ and an implementation I defined as its valuations over the infinite sequence of discrete time steps $(s_j)_{j \in \mathbb{N}_0}$

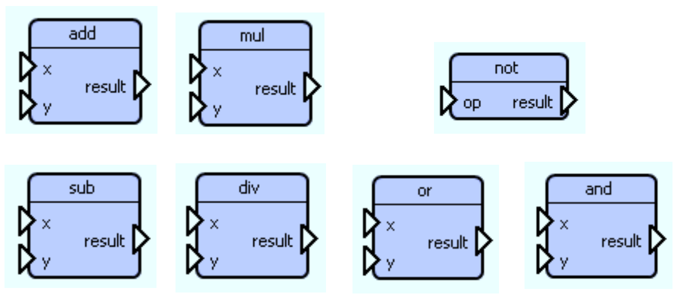
$$out[s_j] := \begin{cases} default & \text{if } j = 0 \\ in[s_{j-1}] & \text{if } j > 0 \end{cases}$$

where $in[s_j]$ indicates the value of the input port at time step s_j and $out[s_j]$ that of the output port, respectively.

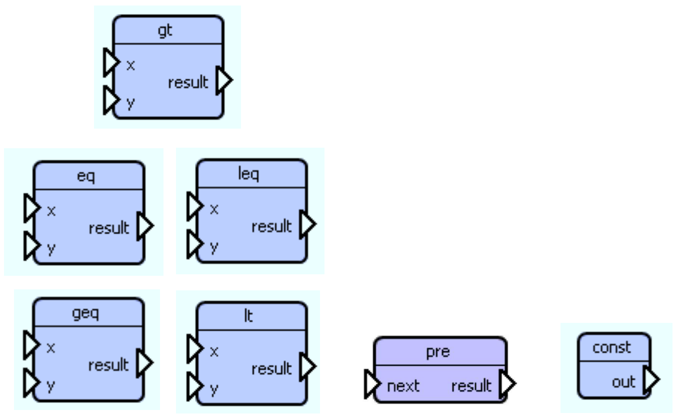
For basic blocks, i. e. functional blocks and timing blocks, the concrete graphical syntax is depicted throughout the Fig(s) 2(a), 2(b), 2(c), and 2(d).

Networks are used to structure the overall system. They are used to provide a high-level system view in order to abstract from implementation details and thus reduce the complexity apparent to the developers. By descending, or decomposing a network, the initial hidden implementation becomes visible. They are realised using so-called *channels* to interconnect a set of units and build up larger data-flow networks. A channel c is a triple $c = \langle n, s, \{d_1, \ldots, d_k\} \rangle$ with n being the identifier and s is the source port which is connected to a set of destination ports $d_i \in \{d_1, \ldots, d_k\}$, with $1 \leq i \leq k$. Together with the included sub-units, networks are defined as: $\langle n, \sigma, \langle U, C \rangle \rangle$ where n is the identifier, σ is the signature as defined for units, and the implementation consists of a set of units U contained in the network together with the set of interconnecting channels C.

The graphical syntax of networks can be learned from the example given in Fig. 1(a).



(a) Arithmetic operators (b) Logical operators



(c) Comparison operators (d) Timing and constant block

Fig. 2. Basic blocks provided by COLA: (a) arithmetical operators, (b) logical operators, (c) comparison operators, and (d) timing block (delay or pre) and the constant block.

Automata are special units, whose implementation is a finite automaton with states and transitions guarded by predicates. Both, states and guards are itself implemented by units: a state's behaviour is defined by a network, the guards are stateless networks, i. e., networks without occurrences of automata and delays since these units are statefull. They have to store their current state in the case of an automaton and their last value in the case of a delay for at least one execution cycle.

Formally, an automaton is a unit $\langle n, \sigma, I\rangle$ with identifier n and signature σ. The implementation of an automaton is given by: $I = \langle Q, q_o, \varDelta\rangle$, where Q is a finite set of state labels, that refer to the names of units, which implement the state's behaviour. Their signature is equal to that of the automaton. q_0 is the name of the initial state and $\varDelta \subseteq Q \times \mathsf{dom}(\mathsf{in}(\sigma)) \times Q$ is the transition relation. $\mathsf{dom}(\mathsf{in}(\sigma))$ is defined as

$$\mathsf{dom}(\mathsf{in}(\sigma)) \overset{def}{=} \mathsf{dom}(\mathsf{type}(a_1)) \times \ldots \times \mathsf{dom}(\mathsf{type}(a_k))$$

where $\mathsf{dom}(\mathsf{type}(a_i))$ denotes the domain of the typed ports $a_i \in \mathsf{in}(\sigma), 1 \leq i \leq k$. $\mathsf{in}(\sigma)$ defines the projection onto the input ports of the signature and respectively $\mathsf{out}(\sigma)$ defines the projection onto the output ports.

Starting from the initial state, the semantics is defined as follows: let q be the current state, if there is an outgoing transition whose guard evaluates to *true*, take it and execute the unit referenced by the target state. If there is no such transition predicate evaluating to *true*, execute the unit referenced by the current state.

An example for the graphical syntax of a COLA automaton is depicted in Fig. 3.

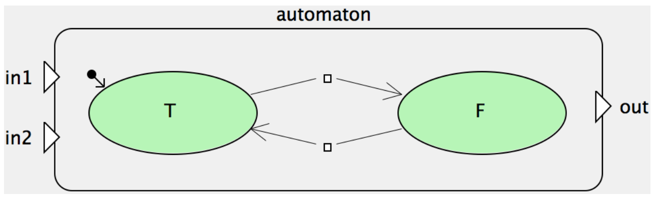


Fig. 3. A COLA automaton with two states T (initial state) and F and from each state a transition to the other one.

4 Translation Schema

In the following, a translation schema from COLA to CPNs is proposed. Therefore, each language construct is translated one after another. Beginning with basic units, namely functional blocks, we give stepwise more and more complex translation schemas for units like networks and automata. We will, however, not introduce translation schemas neither for constant blocks nor for delay units as their translation is straightforward: *constant blocks* – are translated into a single CPN place initialised with the corresponding value, *delays* – for each input and output we generate a separate CPN place as well as a transition to connect them. The input place holds the initialisation value. A translation of a delay, e.g. `pre_1`, can be found in Fig. 8(a). Note that the delay is modelled as a substitution transition. The translation described above is modelled in its subpage (not visible in Fig. 8(a)). Keeping the same structure as the original COLA model should serve for a better understanding of the translation process.

For the translation we use both hierarchical and non-hierarchical CPNs. Units that can be decomposed are translated into hierarchical CPNs, those that cannot into non-hierarchical CPNs. In order to make sure that no value is written into a non-empty place, we define input and output places (and where necessary) as lists of a given data type. Thus, apart from other constraints, each transition connected to such places fires only if its postset is empty. This reflects the behaviour defined in COLA.

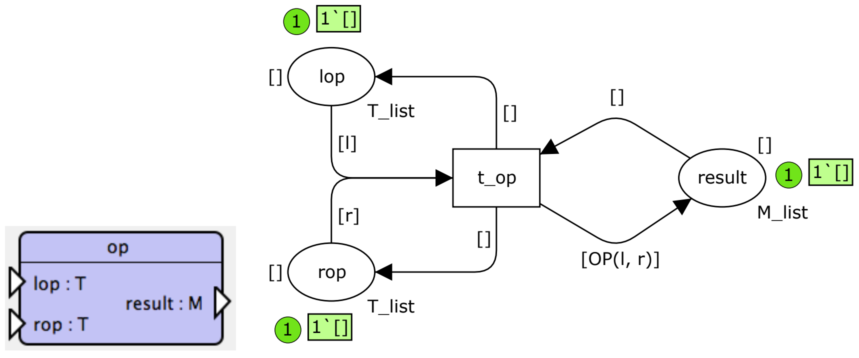


Fig. 4. (a) COLA basic block with two input ports of type T and an output port of type M. (b) Corresponding CPN with three places and one transition.

Before we start with the definition of the translation schema we need first to define a function $\pi : io(\sigma) \rightarrow P$ which maps the set of COLA input and output ports into the set of CPN places, with $\mathsf{io}(\sigma) = in(\sigma) \cup out(\sigma)$.

4.1 Functional Block

In Fig. 4 a COLA functional block and its translation is depicted. The translation schema for a functional block is defined as shown in Fig. 5. Since functional blocks cannot further be decomposed their translation is straightforward. Input and output ports are transformed into CPN places (P), including their corresponding data types (C). A transition (T) is generated to reflect the operation OP and is accordingly connected to places by arcs (A). Arc inscriptions (E) matching the empty list [ ] play a key role for the generated CPN model. On the one hand, they force the transition to fire only if its postset is empty, cf. $a = (result, t_{op})$ in Fig. 4(b). In this way the behaviour defined in COLA is reflected, i. e. no new value is added to an output port unless old values are consumed. On the other hand, they notify other modules connected to them, that the data residing in the input ports has been consumed (cf. the outgoing arcs from t_{op}), i. e. new values can proceed. To achieve this we define lists of used data types (Σ). Variables (V) corresponding to a data type are used to read the input and process the data according to the operation OP, cf. the arc inscription of $a = (t_{op}, result)$ in Fig. 4(b). The guard (G) of the transition is always `true`. All places are initialised (I) with the empty list.

4.2 Network

In COLA networks can consist of a large number of subunits. An example of a network is depicted in Fig. 1. We will describe only the translation of the highest level of a network.The translation schema for COLA data-flow networks

FunctionalBlock

A COLA functional block

$$\mathsf{FB} \;=\; \langle n, \sigma = (\langle lop : tt, rop : tt\rangle \rightarrowtail \langle result : m\rangle), \mathsf{OP}\rangle$$

is translated into a CPN

$$cpn = (P, T, A, \Sigma, V, C, G, E, I)$$

using the following schema:

Schema

$P = \pi(\mathsf{in}(\sigma) \cup \mathsf{out}(\sigma))$, i.e. $\{lop, rop\} \cup \{result\}$

$T = \{t_{op}\}$

$A = \{(lop, t_{op}), (rop, t_{op}), (t_{op}, result), (t_{op}, lop), (t_{op}, rop), (result, t_{op})\}$

$\Sigma = \{tt\_l, m\_l\}$, tt_l and m_l are lists of type tt and m, resp.

$V = \{l : tt, r : tt\}$

$$C(p) = \begin{cases} tt\_l & \text{if } p \in \{lop, rop\} \\ m\_l & \text{if } p \in \{result\} \end{cases}$$

$G(t) = \texttt{TRUE}, \forall\ t\ \in\ T$

$$E(a) = \begin{cases} [l] & \text{if } a = (lop, t_{op}) \\ [r] & \text{if } a = (rop, t_{op}) \\ [\mathsf{OP}(l, r)] & \text{if } a = (t_{op}, result) \\ [\,] & \text{if } a \in \{(t_{op}, lop), (t_{op}, rop), (result, t_{op})\} \end{cases}$$

$I(p) = [\,], \forall p \in P$

Network

A COLA network

$$\mathsf{NET} \;=\; \langle n, \sigma, \langle U, C\rangle\rangle$$

is translated into a hierarchical CPN

$$Hcpn = (S, SN, SA, PN, PT, FS, FT, PP)$$

using the following schema:

Schema

$S = \{CPNnetwork\} \cup S_U$

$SN = \{n_{\mathsf{NET}}\} \cup SN_U$, n_{NET} is the identifier of NET

$SA(SN) = \bigcup_{s \in SN} SA(s)$

$PN = \pi(\mathsf{io}(\sigma)) \cup PN_U$

$$PT(p) = \begin{cases} i/o & \text{if } p \in \pi(\mathsf{io}(\sigma)) \\ PT(PN_U) & \text{if } p \in PN_U \end{cases}$$

$$PA(t) = \begin{cases} \{(in_1@CPNnetwork, in_1@network), & \\ (in_2@CPNnetwork, in_2@network), \ldots, & \\ (out_1@CPNnetwork, out_1@network), \ldots\} & \text{if } t = n_{\mathsf{NET}} \\ PA(SN_U) & \text{if } t \in SN_U \end{cases}$$

$PP = 1`CPNnetwork$

Note: FS and FT are not considered during the translation.

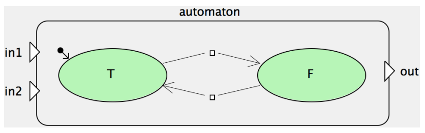
Fig. 5. Functional Block and Network Schema.

is defined as shown in Fig. 5. Each network is translated into a corresponding hierarchical CPN. For the top level of each COLA network a page, the super page, is generated, e. g. *CPNnetwork* in Fig. 1(b). In the following we will refer to the COLA and CPN models and their components in Fig. 1, when necessary to achieve a better understanding of the translation process. The set of other pages, representing the implementation $I = \langle U, C \rangle$ of the network, are included in S_U, where U is the set of subunits participating in the network. Each of these units is separately translated corresponding to its schema type. The translation of the set of channels C is not explicitly given. However, they are important for establishing the connectivity between translated components, e. g. if there is a connection/channel from a COLA unit A to a unit B, in the corresponding CPN model the output places of A are correspondingly glued together with the input places of B (unless some other criteria apply). Each subunit is represented by the set of substitution transitions SN, which consists of a transition, e. g. $n_{\mathsf{NET}} = \mathsf{NETWORK}$, and the set of those ($SN_U$) appearing in the subunits in U. SA maps each substitution transition to their implementations in the subpages, e. g. transition NETWORK to the subpage *network*. The set of input and output nodes of *CPNnetwork* (in_1, $in_2, \ldots$) are unified with those of the subpages PN_U building the set PN. Most of port nodes are of type (PT) *i/o* as described in the schema. Now we just need to define the assignment (PA) of port nodes to socket nodes, e. g. in_1 in *network*, denoted in_1@*network*, is assigned to in_1 in *CPNnetwork* (in_1@*CPNnetwork*). Since the nodes in both pages share commonly the same name, the tuple (out_1@*CPNnetwork*, $result_3\,out_1$@*network*) illustrates best such an assignment.

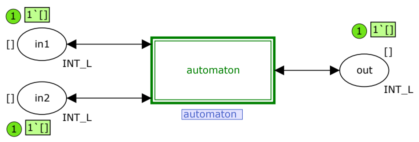
4.3 Automaton

Automata are the most complex units of COLA. Figure 6 shows a COLA automaton and its CPN representation. For the translation of an automaton we introduce a two-step schema (cf. Fig. 6). In the first step we describe the highest abstraction level as a hierarchical CPN. In the second step the functionality of the automaton, i.e. guard evaluation and state switching, is described as a non-hierarchical CPN. For each state of the automaton, e.g. `T` and `F`, there exists a separate transition, which serves as a substitution transition for the implementation of the underlying network unit (cf. Fig. 8(b)). The same figure would represent also the functionality of the automaton in Fig. 6, by only replacing `do_nothing` and `working` with `T` and `F`, respectively. The first translation step is similar to the translation of a network, thus we give no further description. We have, however, to stress that the index Q represents the implementation of the underlying network for each automaton state in Q. The set of their corresponding substitution transitions is denoted Q_T.

The second schema describes by means of non-hierarchical CPNs the next lower level page (*automaton)*. Besides the port nodes, determined by $\pi()$, which are needed to be assigned to sockets of the parent or super page (*CPNautomaton*), there are two additional places *State* and *Activated* added to the set of places P. Place *State* is of type *State* and holds the identifiers of each state



(a) COLA automaton



(b) CPN page *CPNautomaton*

Automaton

A COLA automaton
AUT $= \langle n, \sigma, I\rangle$, $I = \langle Q, q_0, \Delta\rangle$, $\Delta \subseteq Q \times \mathsf{dom}(\mathsf{in}(\sigma)) \times Q$
is translated into a hierarchical CPN

$$Hcpn = (S, SN, SA, PN, PT, FS, FT, PP)$$

using the following schema:

*Schema*1

$S = \{CPNautomaton\} \cup S_Q$

$SN = \{n_{\mathsf{AUT}}\} \cup SN_Q, n_{\mathsf{AUT}}$ is the identifier of AUT

$SA(SN) = \bigcup_{s \in SN} SA(s)$

$PN = \pi(\mathsf{io}(\sigma)) \cup PN_Q$

$$PT(p) = \begin{cases} i/o & \text{if } p \in \pi(\mathsf{io}(\sigma)) \\ PT(PN_Q) & \text{if } p \in (PN_Q) \end{cases}$$

$$PA(t) = \begin{cases} \{(in_1@CPNautomaton, in_1@automaton), \\ (in_2@CPNautomaton, in_2@automaton), \ldots, \\ (out_1@CPNautomaton, out_1@automaton), \ldots\} & \text{if } t = n_{\mathsf{AUT}} \\ PA(SN_Q) & \text{if } t \in (SN_Q) \end{cases}$$

$PP = 1`CPNautomaton$

automaton represents the subpage of n_{AUT}.
Note: FS and FT are not considered during the translation.

*Schema*2 – *page* automaton

$P = \{State, Activated\} \cup \pi(\mathsf{in}(\sigma) \cup \mathsf{out}(\sigma))$

$T = \{activate\ State\} \cup Q_T, A = \{(State, activate\ State), (activate\ State, State), (activate\ State, Activated), (Activated, activate\ State), \ldots\}$

$\Sigma = \{State, State\_L, \} \cup \mathsf{D}$, with $\mathsf{D} = \mathsf{dom}(\mathsf{in}(\sigma))$

$V = \{s : State\} \cup \{v_1 : t_1, \ldots, v_n : t_n\}, t_i \in \mathsf{D}, 1 \leq i \leq n, n = \mid \mathsf{in}(\sigma) \mid$

$$C(p) = \begin{cases} State & \text{if } p = State \\ State\_L & \text{if } p = Activated \\ \mathsf{D} & \text{if } p \in \pi(\mathsf{io}(\sigma)) \end{cases}$$

$G(t) = \texttt{TRUE}, \forall\ t\ \in\ T$

$$E(a) = \begin{cases} s & \text{if } a = (State, activate\ State) \\ state(s, \{v_1, v_2, \ldots\}) & \text{if } a = (activate\ State, State) \\ [state(s, \{v_1, v_2, \ldots\})] & \text{if } a = (activate\ State, Activated) \\ \ldots \end{cases}$$

$I(State) = q_0$

Fig. 6. Exemplary (a) COLA automaton with two states and (b) its translation, and the translation schema.

in Q. *Activated* holds the currently active state. The transition *activateState* is responsible for the initialisation of state switching, by feeding the function *state()* with input data and the actual active state. The purpose of function *state()* is to check and control the switching between states, according to the defined guards of the automaton. How this works has already been described in Sect. 3.1. Let $G = \{g_1, g_2, \ldots, g_n\}$ be the set of the guards of an automaton, $V = \{v_1, v_2, \ldots, v_m\}$ the set of variables used in the guards and $S = \{s_1, s_2, \ldots, s_i\}$ the set of states of the automaton, with $s \in S$. We define the *state*() function and the colour sets *State* and *State_L* as follows:

```
fun state(s, v_1, ..., v_m) = if s = s_1 andalso g_1 then s_2
                              else
                              if s = s_2 andalso g_2 then s_3
                              ...
                              else s;
colset State   = with s_1 | s_2 | ... | s_i;
colset State_L = list State;
```

The rest of the translation schema is straightforward.

4.4 Translation Algorithm

The idea of the outlined Algorithm 1 is to translate COLA models into CPNs in a DFS manner. For each visited unit, the corresponding translation schema is applied. Once all units are translated there will be loose components and a superfluous number of places (representing each input and output port of each component). To reduce the number of places and establish the corresponding connectivity between components, we glue together input and output places (cf. line 19) regarding the defined channels in the original COLA model, i. e. the corresponding source and destination ports. Finally, to accommodate the structure of the generated hierarchical CPN, the connection between subpages and their parent pages is established by assigning ports to sockets (cf. line 20).

There are two special cases that need to be considered during the translation of a network: first, if multiple ports read from one and the same port (cf. port *out* of the constant block in Fig. 1(a)). In this case, we translate the connection in that way that the source of the channel is translated to as many places as there are destinations (cf. places $y_1 out_3$ and $x_2 out_3$ in Fig. 1(c)). Second, the input and output of a unit are not connected (cf. Fig. 7 the implementation of the `do_nothing` state). Therefore, we create a *new* place and connect it with the *input* transition and other transitions accordingly (cf. Fig. 8(c)). This is done to make sure that the data flow in the network is not broken, i. e. we want to establish a correct consumption of the input data in order to proceed to the output, as required in COLA.

Note that one can merge the transitions *input* and *out*, thus not needing to add the *new* place at all. The transition *input* can often get merged with other transitions and reduce the size of the net, e. g. one could merge *input* and

Algorithm 1: COLA2CPN

```
Data: COLA model
Result: CPN model
while (not all units $u \in U$ have been visited) do
    perform a DFS traversal on the COLA model;
    switch (u instanceof) do
        case (functional block)
            if (u isA constant) then
                create a single place $p$;
                initialise $p$ accordingly;
            else
                FunctionalBlock($u$);
        case (network)
            Network($u$);
            create a transition input to collect the incoming data;
            connect input according to the connections in $u$ (channels);
        case (automaton)
            Automaton($u$);
        case (delay)
            Delay($u$);
            initialise the translation
glue input and output places together, according to their connectivity in the
COLA model;
assign ports to sockets;
```

add (cf. Fig. 8(d)) and deleting the places *in_1* and *in_2*, without changing the behaviour of the net.

5 Example

In Fig. 7, a screenshot of the COLA simulation tool is depicted. It shows a high level COLA system consisting basically of two automata, two input constants and two delay operators (pre). Each automaton has two states, namely do_nothing and working. In both cases, do_nothing always provides the value 0 as output, concerning the behaviour of working, however, both automata show a different implementation. automaton_1 performs the subtraction of the values present at the input ports in_1 and in_2 (out := in_1 - in_2). The state working of automaton_2 increases the input value at port in_1 by 3 (out := in_1 + 3). The modelled transition relations are omitted for the sake of clarity.

In COLA a deadlock in a classical sense is not possible. This is due to the fact that the COLA semantics dictates that at each tick of the system execution a new value is assigned to each output port. A deadlock from a Petri net point of view is compared best with a COLA system, that is stuck in an automaton state, which cannot be changed anymore. This might, however, be a system design decision. But in many cases, as in the given example, it is a modelling error. Regarding the example, the values present at the output ports result of both delays have a special behaviour: at the first tick both ports emit the value 1. To simplify matters, we write these values as a result vector $\mathbf{r} = \left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)$ where the upper value corresponds to the port value of the upper delay, and the lower value to the lower delay, respectively. Both values have been set by the developer as default values for the delays. When considering the behaviour over time we use a matrix-like notation, i. e., the ith column of the matrix represents the output values after the ith tick: $\mathbf{M}^\infty = \left(\begin{smallmatrix}1 & 2 & -3 & -4 & 0 & 0 \dots\\1 & 6 & 7 & 0 & 0 & 0 \dots\end{smallmatrix}\right)$. For this simple example, the following infinite sequence

$$\mathbf{M}^\infty = \begin{pmatrix}1 & 2 & -3 & -4\\1 & 6 & 7 & 0\end{pmatrix} \circ \begin{pmatrix}0\\0\end{pmatrix}^\omega$$

of port valuations is obtained, i. e., after a finite number of steps (four in this case) the system reaches a deadlock-like state and from then on only emits $\mathbf{r} = \left(\begin{smallmatrix}0\\0\end{smallmatrix}\right)$ as result. However, for more complex examples, similar behaviour cannot be detected by the developer by solely using the COLA simulator. Here, the power of the CPN Tools becomes important.

After translating the COLA model into a CPN, using the outlined translation algorithm, the CPNs depicted in Fig. 8 are obtained. The idea is to automatically construct the state space of the CPN models and finally create the *state space report* which contains information about standard behaviour properties: dead markings, dead and live transitions, etc. These information collected in the report support the analysis of a system in an early stage of its development and help to decreases the number of design errors. Furthermore, one can check other specific behavioural properties by using predefined *query functions* provided by CPN Tools to write user-defined analysis algorithms. For our example, the CPN Tools reported a set of live transitions shown as an excerpt of the report below.

```
Live Transition Instances
--------------------------------
    automaton1'activate State 1
    automaton2'activate State 1
    doNothing1'input 1
    doNothing1'out 1
    doNothing2'input 1
    doNothing2'out 1
    pre1'delay 1
    pre1'init 1
    pre2'delay 1
    pre2'init 1
```

The expected behaviour is reflected in this result to the effect that the transitions representing both working states are not contained. That means for the CPN, that there is a marking from which there exists no path containing these

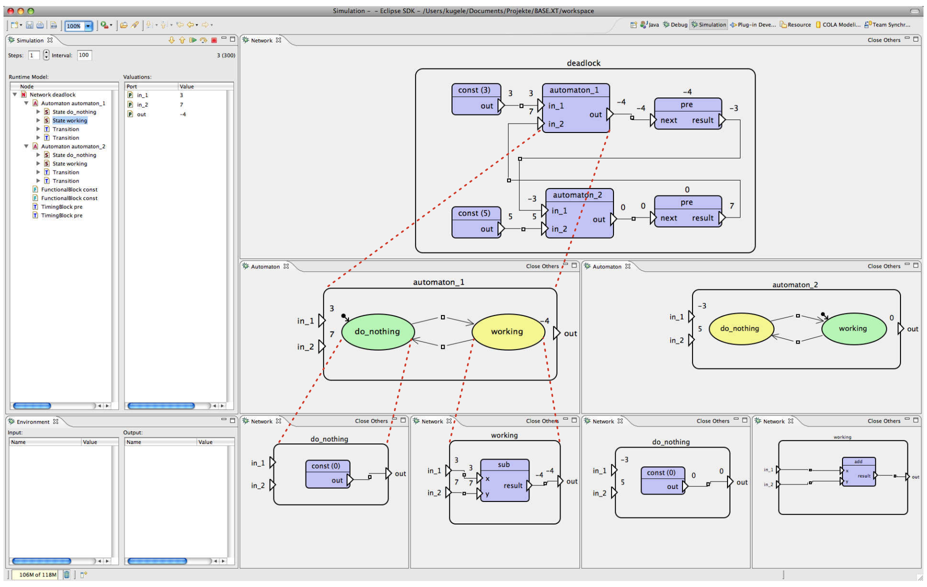
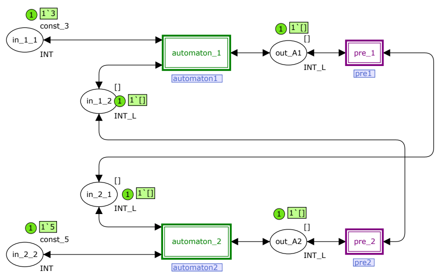
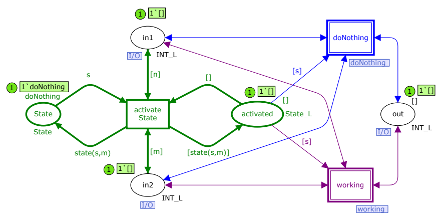


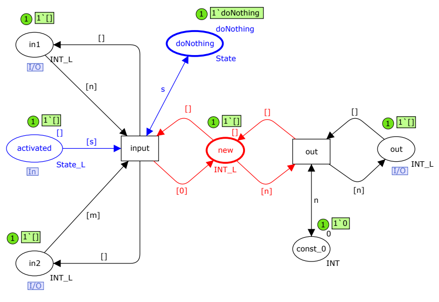
Fig. 7. COLA Simulator: Dashed lines are added manually to clarify the hierarchical decomposition.



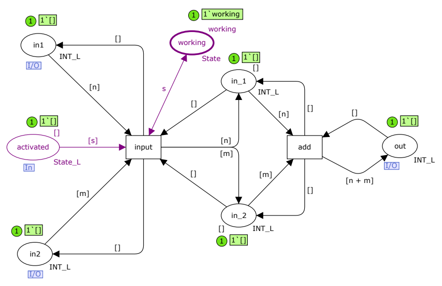
(a) CPN example



(b) Automaton



(c) State *do nothing*



(d) State *working*

Fig. 8. CPN example: (a) The highest abstraction level of the CPN example. (b) Realisation of an automaton. (c) Realisation of the state *do nothing*. (d) Realisation of the state *working* (*automaton_2*).

transitions. In other words—from a COLA point of view—it is possible to reach a system state that prohibits a change to a distinguished system state (working in our case). Based on this information, the developer has to check whether the modelled system behaviour is what was desired. If this it not the case, a modelling error has been detected. This is only one of the many observation one can receive, i. e. by far not all what an analysis process can yield. The intention of the analysis example was however to show how helpful can be such analysis result. Being beyond the scope of this paper, the analysis of COLA models will not be discussed.

An issue, however, remains the state space explosion problem. To alleviate it a number of state space reduction methods (symmetry, equivalence, sweep line) have been developed and integrated into CPN Tools. Furthermore, Khomenko et al. [35] presented an improvement of the *unfolding* technique which can be applied to all classes of high-level Petri nets. Based on this work, in [36] a prototype has been proposed and developed for unfolding a subclass of n-safe CPNs. In the ASCoVeCO project [37] a platform (ASAP) is being developed aiming for the integration of various analysis methods into one environment, as well as giving the possibility to extend the existing tool collection, thus increasing the analysis possibilities for CPNs.

Our translation concept can only profit from such an analysis environment, facilitating broader analysis aspects for COLA models in return.

6 Conclusions

In this paper we introduced a mathematically sound schema for the translation of the synchronous data-flow language COLA into Coloured Petri Nets. This translation schema allows the combining of the strengths of both modelling techniques to have a powerful model analysis methodology at hand. The toy example presented here showed the applicability of the presented approach by providing hints for possible design errors. A possible future extension could be the use of Timed Coloured Petri Nets to fit better the synchronous paradigm that COLA follows. Furthermore, we want to facilitate the automatically translation of COLA models into CPNs. This will allow us to deal with and analyse more interesting, larger and real-life systems modelled in COLA.

We believe that this approach is feasible to be also applied to other synchronous data-flow languages, like Lustre for instance.

Acknowledgments

We would like to thank Andreas Holzer and the anonymous reviewers for their fruitful comments in finalising this paper.

References

1. Broy, M.: Automotive software and systems engineering (panel). In: MEMOCODE. (2005) 143–149

2. The MathWorks Inc.: Using Simulink. (2000)
3. Berry, G., Gonthier, G.: The esterel synchronous programming language: design, semantics, implementation. Sci. Comput. Program. **19**(2) (1992) 87–152
4. Tripakis, S., Sofronis, C., Caspi, P., Curic, A.: Translating discrete-time simulink to lustre. Trans. on Embedded Computing Sys. **4**(4) (2005) 779–818
5. Kugele, S., Tautschnig, M., Bauer, A., Schallhart, C., Merenda, S., Haberl, W., Kühnel, C., Müller, F., Wang, Z., Wild, D., Rittmann, S., Wechs, M.: COLA – The component language. Technical Report TUM-I0714, Institut für Informatik, Technische Universität München (September 2007)
6. Kugele, S., Haberl, W.: Mapping Data-Flow Dependencies onto Distributed Embedded Systems. In: Proceedings of the 2008 International Conference on Software Engineering Research & Practice, SERP 2008, Las Vegas, Nevada, USA (July 2008)
7. Wang, Z., Haberl, W., Kugele, S., Tautschnig, M.: Automatic Generation of SystemC Models from Component-based Designs for Early Design Validation and Performance Analysis. In: Proceedings of the 7th International Workshop on Software and Performance, WOSP 2008, Princeton, NJ, USA, ACM (June 2008) 23–26
8. Kugele, S., Haberl, W., Tautschnig, M., Wechs, M.: Optimizing automatic deployment using non-functional requirement annotations. In Margaria, T., Steffen, B., eds.: Leveraging Applications of Formal Methods, Verification and Validation. Volume 17 of CCIS., Springer (2008) 400–414
9. Haberl, W., Tautschnig, M., Baumgarten, U.: Running COLA on Embedded Systems. In: Proceedings of The International MultiConference of Engineers and Computer Scientists 2008. (March 2008)
10. Herrmannsdoerfer, M., Haberl, W., Baumgarten, U.: Model-level Simulation for COLA. In: International Workshop on Modeling in Software Engineering (MISE'09: ICSE Workshop 2009). (2009)
11. Haberl, W., Birke, J., Baumgarten, U.: A Middleware for Model-Based Embedded Systems. In: Proceedings of the 2008 International Conference on Embedded Systems and Applications, ESA 2008, Las Vegas, Nevada, USA (July 2008)
12. Haberl, W., Tautschnig, M., Baumgarten, U.: From COLA Models to Distributed Embedded Systems Code. IAENG International Journal of Computer Science **35**(3) (September 2008) 427–437
13. Kühnel, C., Bauer, A., Tautschnig, M.: Compatibility and reuse in component-based systems via type and unit inference. In: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), IEEE Computer Society Press (2007)
14. Wang, Z., Sanchez, A., Herkersdorf, A.: Scisim: a software performance estimation framework using source code instrumentation. In: Proceedings of the 7th international workshop on Software and performance (WOSP '08), New York, NY, USA, ACM (2008) 33–42
15. Jensen, K.: Coloured Petri nets (2nd ed.): basic concepts, analysis methods and practical use, volume 1. Springer-Verlag, London, UK (1996)
16. Jensen, K.: Coloured Petri nets: basic concepts, analysis methods and practical use, volume 2. Springer-Verlag, London, UK (1997)
17. Jensen, K.: Coloured Petri nets: basic concepts, analysis methods and practical use, volume 3. Springer-Verlag New York, Inc., New York, NY, USA (1997)
18. Reisig, W.: Petri nets: an introduction. Springer-Verlag New York, Inc., New York, NY, USA (1985)
19. Paulson, L.C.: ML for the working programmer (2nd ed.). Cambridge University Press, New York, NY, USA (1996)

20. Standard ML. `http://www.standardml.org/`
21. CPN Tools. `http://www.daimi.au.dk/CPNTools/`
22. Kristensen, L.M., Jensen, K.: Specification and validation of an edge router discovery protocol for mobile ad hoc networks. In: SoftSpez Final Report. (2004) 248–269
23. Kristensen, L.M., Billington, J., Qureshi, Z.: Modelling military airborne mission systems for functional analysis. (2001)
24. Petrucci, L., Billington, J., Kristensen, L.M., Qureshi, Z.H.: Developing a formal specification for the mission system of a maritime surveillance aircraft. In: ACSD '03: Proceedings of the Third International Conference on Application of Concurrency to System Design, Washington, DC, USA, IEEE Computer Society (June 2003) 92–101
25. Qureshi, Z.H.: Formal modelling and analysis of mission-critical software in military avionics systems. In: SCS '06: Proceedings of the eleventh Australian workshop on Safety critical systems and software, Darlinghurst, Australia, Australia, Australian Computer Society, Inc. (2006) 67–77
26. Gottschalk, F., van der Aalst, W.M.P., Jansen-Vullers, M.H., Verbeek, H.M.W.: Protos2cpn: using colored petri nets for configuring and testing business processes. STTT **10**(1) (2008) 95–110
27. Kang, H., Yang, X., Yuan, S.: Modeling and verification of web services composition based on cpn. In: NPC '07: Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops, Washington, DC, USA, IEEE Computer Society (2007) 613–617
28. Yang, Y., Tan, Q., Xiao, Y., Liu, F., Yu, J.: Transform bpel workflow into hierarchical cp-nets to make tool support for verification. In: APWeb. (2006) 275–284
29. Hinz, S., Schmidt, K., Stahl, C.: Transforming bpel to petri nets. In: Business Process Management. (2005) 220–235
30. Fernandes, J.M., Tjell, S., Jorgensen, J.B., Ribeiro, O.: Designing tool support for translating use cases and uml 2.0 sequence diagrams into a coloured petri net. In: SCESM '07: Proceedings of the Sixth International Workshop on Scenarios and State Machines, Washington, DC, USA, IEEE Computer Society (2007) 2
31. Amorim, L., Maciel, P.R.M., Jr., M.N.N., Barreto, R.S., Tavares, E.: Mapping live sequence chart to coloured petri nets for analysis and verification of embedded systems. ACM SIGSOFT Software Engineering Notes **31**(3) (2006) 1–25
32. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley (1998)
33. Bauer, A., Broy, M., Romberg, J., Schätz, B., Braun, P., Freund, U., Mata, N., Sandner, R., Ziegenbein, D.: AutoMoDe — Notations, Methods, and Tools for Model-Based Development of Automotive Software. In: Proceedings of the SAE 2005 World Congress, Detroit, MI, Society of Automotive Engineers (April 2005)
34. Maraninchi, F., Rémond, Y.: Mode-automata: a new domain-specific construct for the development of safe critical systems. Science of Computer Programming **46**(3) (2003) 219–254
35. Khomenko, V., Koutny, M.: Branching processes of high-level petri nets. In Garavel, H., Hatcliff, J., eds.: TACAS. Volume 2619 of Lecture Notes in Computer Science., Springer (2003) 458–472
36. Januzaj, V.: CPNunf: A tool for McMillan's Unfolding of Coloured Petri Nets. In: Proceedings of 8th Workshop on Practical use of Coloured Petri Nets and the CPN Tools. (2007) 147–166
37. The ASCoVeCO project. `http://www.daimi.au.dk/~ascoveco/`