Translating Colored Control Flow Nets into Readable Java via Annotated Java Workflow Nets

Kristian Bisgaard Lassen and Simon Tjell

Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
{k.b.lassen,tjell}@daimi.au.dk

Abstract. In this paper, we present a method for developing Java applications from Colored Control Flow Nets (CCFNs), which is a special kind of Colored Petri Nets (CPNs) that we introduce. CCFN makes an explicit distinction between the representation of: The system, the environment of the system, and the interface between the system and the environment. Our translation maps CCFNs into Annotated Java Workflow Nets (AJWNs) as an intermediate step, and these AJWNs are finally mapped to Java. CCFN is intended to enforce the modeler to describe the system in an imperative manner which makes the subsequent translation to Java easier to define. The translation to Java preserves data dependencies and control-flow aspects of the source CCFN. This paper contributes to the model-driven software development paradigm, by showing how to model a system, environment, and their interface, as a CCFN and presenting a fully automatic translation of CCFNs to readable Java code.

1 Introduction

In this paper, we document an approach to automatic generation of executable Java programs based on Colored Petri Net [13, 14] (CPN) models with a clear distinction between the environment and system domains [19]. We will define these restricted CPNs as Colored Control Flow Nets (CCFNs).

The approach that this paper presents is intended to contribute to the model-driven software paradigm by allowing the user to build, simulate, and analyze CCFN models in tools such as CPN Tools [9, 14], and later to automatically translate the CCFN models into Java code. The purpose of the code generation we present in this paper is not to produce the end product - i.e. the final implementation - but rather to take a step in the direction of tool-based support for generating rapid prototypes.

CPN models are useful for expressing functional requirements for reactive systems by representing required behavior at a high level of abstraction [10, 11]. Reactive systems [20] are a special class of computer systems that are characterized by a close coupling with the surrounding environment through an interface of sensors and actuators. Typical examples of reactive systems include vending machines, elevator controllers, and cruise controllers.

When a reactive system is modeled it is also necessary to model the parts of the environment interacting with the reactive system. The reactive system interacts in a predictive manner with its environment by exchanging observable events and state changes. The reason why it is generally necessary to model both the system and the environment is that a model of the system alone would be inactive without the simulation of incoming stimuli from the environment to which the system can react. This property is common to all reactive systems: No spontaneous behavior is exhibited. When we use the models for abstract representation of behavior, we want to be able to execute scenarios of interaction between the environment and system domains and this is why both domains must be represented in the model.

Our intention is that the translation generates readable Java. By readable, we mean that the generated Java should be readable in the sense that the translation builds a Java program, where behavioral constructs such as sequence, choice, or parallelism, are implemented in the same way as a programmer would have done. It should therefore be more easy to understand, edit, and maintain the generated Java code.

We do not map CCFNs directly to Java, but instead map CCFNs to Annotated Java Workflow Nets (AJWN) (we define AJWN later in this paper), and map AJWNs to Java as shown by the translations *T0*

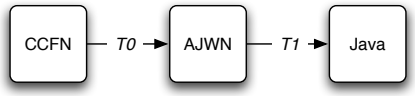


Fig. 1. Two-phase approach for mapping CCFN to Java.

and *T1* in Figure 1. *T0* maps the control-flow of the CCFN to a AJWN, which is essentially a Workflow net [1] (WF-net) where each transition is associated with a Java statement. The focus of this step is to map construct in the CCFN to Java statements, since the translation of the control-flow is straight-forward by the way CCFN is defined. *T1* focus on mapping various behavioral constructs in the AJWN control-flow to a Java code. We found that this division of concerns made the translation more smooth and extensible, than if we had mapped CCFN directly to Java.

Our definition of CCFN will encourage and force the modeler to build models that behave like imperative programs. Transitions will correspond to statements and what the statements can do is to route the control-flow, and either update the state by setting the value of a variable, read an event, or write an event. In this paper, when we refer to the imperative paradigm, we talk about a situation where only those statement types are possible and control-flow is composed of statements.

Besides making a translation from CCFN to Java we have made a translation that is extensible with respect to the collection of Java statements and Java control-flow constructs you want to be able to map to. In this paper, we have chosen a sensible subset of Java to map to, but it is possible to extend either *T0* or *T1* to handle an even bigger subset of Java statements and control-flow constructs, as we discuss throughout paper.

The paper is structured as follows: in Section 2 we introduce and define the class of CPNs called CCFNs that we use for modeling reactive systems and their environments; Section 3 presents the intermediate language AJWN used in the translation; Sections 4 and 5 describe and define the translation algorithm respectively from CCFNs to AJWNs and AJWNs to Java code; in Sections 6 we discuss related work; and finally in Section 7 we conclude and present future work.

2 Modeling Reactive Systems with CPNs

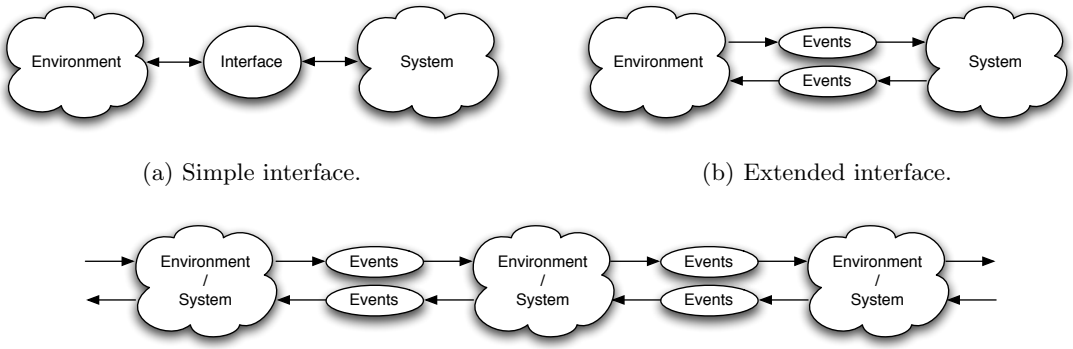
In this section we describe how we model reactive systems, and how we represent these using CPNs. In Section 2.1 we motivate and explain how we wish to model reactive systems using CPNs and in Section 2.2 we introduce CCFNs that we will use for modeling reactive systems.

2.1 Colored Petri Nets and Reactive Systems

In Figure 2(a) we show a simple way to model reactive systems in CPNs. There are basically three parts: An environment modeled; a system; and finally some interface modeled as places. In the Figure 2(a) the interface is simply a single place where the tokens represent messages. In reactive systems there is often a clear distinction between events sent to the system and events sent to the environment. This distinction is reflected in the model by dividing the interface place in Figure 2(a) into two places that model each of these aspects as shown in Figure 2(b). In this paper, we will use CCFN to model both the environment and system, so that the environment and its interface is a CCFN model and the system and its interface is a CCFN model.

We generalize the idea of Figure 2(b) to include more than one kind of environment and system, by adding more substitution transitions; see Figure 2(c). This generalization allows a part of a model to represent both a system and an environment - i.e. the model of an environment may itself represent a system connected to yet another environment and so forth. In our approach we allow the modeler to model a reactive system as shown in Figure 2(c), and we are then able to translate each model of environment/system into Java code.

It should be noted that while we require that the parts of a model that are to be translated into Java code should be expressed by means of CCFN, it is still possible to express other parts using the full CPN language. For example, it would be possible to express the environment domain as a traditional CPN model and the system as a CCFN model. The only requirement in this context is that the interaction between these two domains should be performed through interface places as specified by CCFN.



(a) Simple interface.

(b) Extended interface.

(c) Generalization of Figure 2(b).

Fig. 2. Different ways to model reactive systems using CPNs.

2.2 Colored Control Flow Nets

CCFNs is designed to facilitate two design aspects: (1) Reactive systems support, as well as (2) support for building models in an imperative fashion. The reactive system design support that CCFNs offer is the same as what is presented in Figure 2(c), i.e. we allow the user to model multiple environments/systems and link them together using places to model events going in and out of the environment/system.

The reason why the second aspect of building imperative models is so important is that it makes the translation from CCFN to readable Java code (Java is imperative) more manageable. If we did not consider this aspect, we would have to map CPNs that contain non-imperative constructions to imperative constructions in Java, which is outside the scope of our research. It is through our definition of CCFNs, which we will come to shortly, that we allow the modeler to model reactive systems, while we enforce that he use an imperative style of modeling.

Before we give a formal definition of CCFN, let us look at the small example in Figure 3; notice it is not important to know exactly what the arc inscriptions mean. The CCFN models a small process that will read an event "input" and assign the value associated with "input" to the variable y; do an operation square on y and assign the result to x; and finally, generate an "output" event associated with the value of x until x>=10. When x reaches this value the process transition Quit becomes enabled and Receive disabled. The access to the variable x is performed through a function (get) that returns the value of a specific variable found in a STATE value. Each CCFN has a single place with the color set STATE and we will simply refer to this place as the *state place*. Two examples of operators lt ($<$) and gte ($\geq$) are used to evaluate the value of the variable in the guards. Later we will see that we distinguish between five types of transitions: Enter process and Leave process are a special kinds of transitions that are used to set up the CCFN so that it has single entry and exit points; Operation and Quit we call *state update* transitions, since they access the state place; Receive is called a *read event* transition since it reads a value from an IN_EVENT place and write the result to the state place; Send is a *write event* transition since it reads a value from the state place and puts it on a OUT_EVENT place; and, finally there is a transition type that is not in this example, and it is

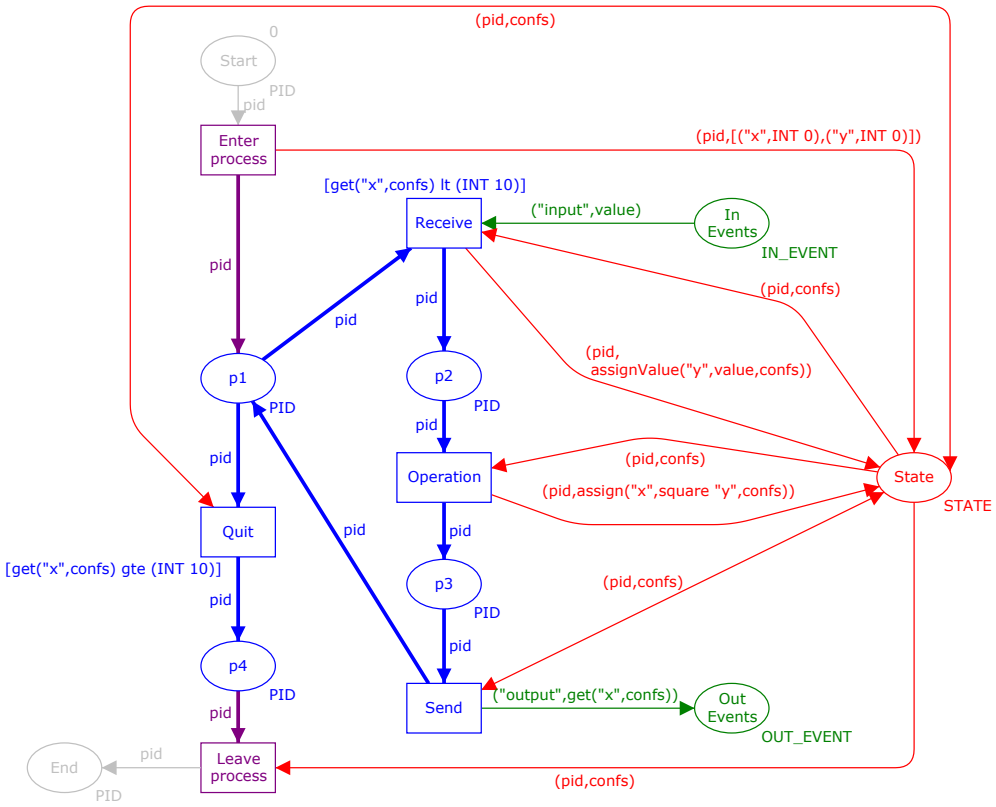


Fig. 3. Small example of a CCFN.

called a silent step transition, and it simply a place that is only connected to places of type PID. silent step transition do not update the state place or read or write events, it is simply a silent step in the execution of the process; hence the name.

Before we define CCFN, we first introduce the allowed place types in a CCFN in Definition 1, and later the allowed arc inscriptions in Definition 2.

Definition 1 (CCFN place types). *A place may either have the color set type* PID *(a process id),* ***STATE****,* ***IN_EVENT****, or* ***OUT_EVENT****, which are defined as follows:*

colset PID = INTEGER
colset STATE = product PID * CONFIGURATION_LIST
colset IN_EVENT = CONFIGURATION
colset OUT_EVENT = CONFIGURATION

where

colset IDENTIFIER = STRING
colset VALUE = union STRING:STRING + INT:INT + BOOL:BOOL + VOID
colset CONFIGURATION = product IDENTIFIER * VALUE
colset CONFIGURATION_LIST = list CONFIGURATION

Notice that CONFIGURATION_LIST is similar to the concept of *environment* in a program execution, however, we choose not to use this term, in order to avoid confusion with the term *environment* used in the context of reactive systems. Another important point is that it is possible to define two different variables

with the same identifier in a CONFIGURATION_LIST. It is up to modeler to ensure that this does not happen, either by hand or by formal analysis. The intentions of the place types are the following: PID will be used to model the control flow of a CCFN; STATE is for modeling the state of a process - i.e. the state of either the system or the environment - expressed in CCFN; IN_EVENT and OUT_EVENT are used to model incoming and outgoing events for the respective CCFN.

Definition 1 states that a place can only contain a process identifier, which is simply an integer; a state, which is a pair that binds a process identifier to a configuration; or, either an input or output event that is represented as a pair of identifier and value. Examples of process identifiers are 0, 1, 2, ...; states are (29, [("x", INT(399)), ("y", BOOL(false)), ("z", STRING("foo"))]), meaning that the process with identifier 29 is in a configuration list with three declared variables x, y, and z that are assigned values integer 399, boolean false, and string "foo"; and finally, an input or output event ("event1",VOID) for the event "event1" and no value, and ("event2",INT 4) for the event "event2" with the integer value 4.

In the following, we will write [X] to denote any expression where Type([X]) = X, $\mathbb{B}$ the boolean type, and $\mathbb{S}$ for the string type. We are now ready to introduce the allowed arc inscriptions in a CCFN. The definition will be followed by a more detailed description of its elements.

Definition 2 (Allowed arc inscriptions in CCFNs). *The only allowed arc inscriptions in a CCFN are*

(i) pid *(used to describe a process id); color set* PID.
(ii) (pid,confs) *(used to describe a process and its configuration confs); color set* STATE.
(iii) (pid,assign([IDENTIFIER],function,confs)) *(a pid and an* assign *expression that evaluates to an updated configuration, where the variable specified by the identifier is assigned the result of calling the function); color set* STATE. *If variable is the empty string the result of the evaluation is not assigned to anything in the configuration.*
(iv) (pid,assignValue([IDENTIFIER],[VALUE],confs)) *(a variation of (iii) that simply assign the variable specified by the identifier to the value in the configuration).*
(v) ([IDENTIFIER],value) *(an identifier for a named event, and value has type* VALUE*); color set* EVENT.
(vi) ([IDENTIFIER],[VALUE]) *(an identifier for a named event, and an expression with type* VALUE*); color set* EVENT.
(vii) ([IDENTIFIER],get([IDENTIFIER],confs) *(an identifier for a named event, get is a function that returns that value of the variable specified by an identifier); color set* EVENT.

The reason why it is necessary to restrict the allowed arc inscriptions is to ease the translation of the CCFN as we will see later. Another consequence of the definition is that we only allow the modeler to update the token values by using imperative constructions, such as assign, assignValue (a special case of assign), and get.

The signature of the function assign is IDENTIFIER * (α → CONFIGURATION_LIST → VALUE) * CONFIGURATION_LIST → CONFIGURATION_LIST; i.e. assign takes a triple: the name of a variable; a function that when applied to a value with type α and a configuration list, then calculates a value; and, the current configuration list, so that assign returns the updated configuration. The signature of the function assignValue is IDENTIFIER * VALUE * CONFIGURATION_LIST → CONFIGURATION_LIST. The signature of get is IDENTIFIER * CONFIGURATION_LIST → VALUE). Examples of arc inscriptions are: pid, (pid,confs), assign("x",negate,confs), ("button pressed",[("isOn",BOOLEAN(false))]), and get("x",confs).

We are now ready to define CCFNs in Definition 3. For the sake of readability, we have omitted the arguments to get, assign, assignValue where used. See Definition 2 for a definition of these. In this paper, we refer to the preset and preset of a node $x \in P \cup T$ as $\bullet x = \{y \in P \cup T | (y, x) \in N(A)\}$ and $x\bullet = \{y \in P \cup T | (x, y) \in N(A)\}$.

Definition 3 (Colored Control Flow Nets (CCFNs)). *A non-hierarchical Colored Petri Net is a tuple* $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ *satisfying [13, Definition 2.5]. CPN is a Colored Control Flow Net (CCFN) iff*

(i) *(Color sets)* $\Sigma = \{PID, STATE, IN\_EVENT, OUT\_EVENT\}$ *. These color sets are defined in Definition 1.*

(ii) (Places) $P = \{p_{start}, p_{end}\} \cup P_{pid} \cup \{p_{state}\} \cup P_{in\_event} \cup P_{out\_event}$, *where*
- $\forall A, B \in \{\{p_{start}, p_{end}\}, P_{pid}, \{p_{state}\}, P_{in\_event}, P_{out\_event}\} : A \neq B \Rightarrow A \cap B = \emptyset$.
- $|P_{pid}| \geq 1$.

(iii) (Transitions) $T = \{t_{start}, t_{end}\} \cup T_{regular}$, *where*
- $\{t_{start}, t_{end}\} \cap T_{regular} = \emptyset$.

(iv) (Arcs and node function) $N(A) = \{(p_{start}, t_{start}), (t_{start}, p_{state}), (p_{state}, t_{end}), (t_{end}, p_{end})\} \cup F$, *where*
- $\{(p_{start}, t_{start}), (t_{start}, p_{state}), (p_{state}, t_{end}), (t_{end}, p_{end})\} \cap F = \emptyset$.
- $\forall a_1, a_2 \in A : a_1 \neq a_2 \Rightarrow N(a_1) \neq N(a_2)$ *(N is injective).*
- $\forall (x, y) \in F : x \notin \{p_{start}, t_{end}, p_{end}\} \wedge y \notin \{p_{start}, t_{start}, p_{end}\}$.
- $\forall (p_{state}, t) \in F : (t, p_{state}) \in F$.
- $\forall p_{in} \in P_{in\_events}, \forall p_{out} \in P_{out\_events}, \nexists t \in T : (p_{in}, t), (t, p_{out}) \in F$.

(v) (Color function)
$$C(p) = \begin{cases} PID & \textbf{if } p \in \{p_{start}, p_{end}\} \cup P_{pid} \\ STATE & \textbf{if } p = p_{state} \\ IN\_EVENT & \textbf{if } p \in P_{in\_event} \\ OUT\_EVENT & \textbf{if } p \in P_{out\_event} \end{cases}$$

(vi) (Guard function)
$$G(t) = \begin{cases} get\ R\ v,\ R\ is\ a\ boolean\ operator \wedge Type(v) = VALUE & \textbf{if } \bullet t = \{p\} \wedge |p \bullet| > 1 \\ true & otherwise \end{cases}$$

(vii) (Arc expression function) In the restriction of E we assume that pid and confs are declared variables where Type(pid) = PID, Type(confs) = CONFIGURATION_LIST, and Type(value) = VALUE.

$$E(a) = \begin{cases} pid & \textbf{if } N(a) \in P' \times T \cup T \times P', Type(P') = PID \\ (pid, confs) & \textbf{if } N(a) = (p_{state}, t_{end}) \vee N(a) = (t, p_{state}), t \in T_{regular} \\ (pid, [CONFIGURATION]) & \textbf{if } N(a) = (t_{start}, p_{state}) \\ (pid, assignValue) & \textbf{if } N(a) = (t, p_{state}) \wedge (p, t) \in P_{in\_event} \times T_{regular} \\ (pid, assign) & \textbf{if } N(a) = (t, p_{state}) \\ (\mathbb{S}, value) & \textbf{if } N(a) \in P_{in\_event} \times T_{regular} \\ (\mathbb{S}, [VALUE]) & \textbf{if } N(a) \in T_{regular} \times P_{out\_event} \\ (\mathbb{S}, get) & \textbf{if } N(a) \in T_{regular} \times P_{out\_event} \end{cases}$$

(viii) (Initialization function)
$$I(p) = \begin{cases} [PID_{ms}] & , p = p_{start} \\ \emptyset & , otherwise \end{cases}$$

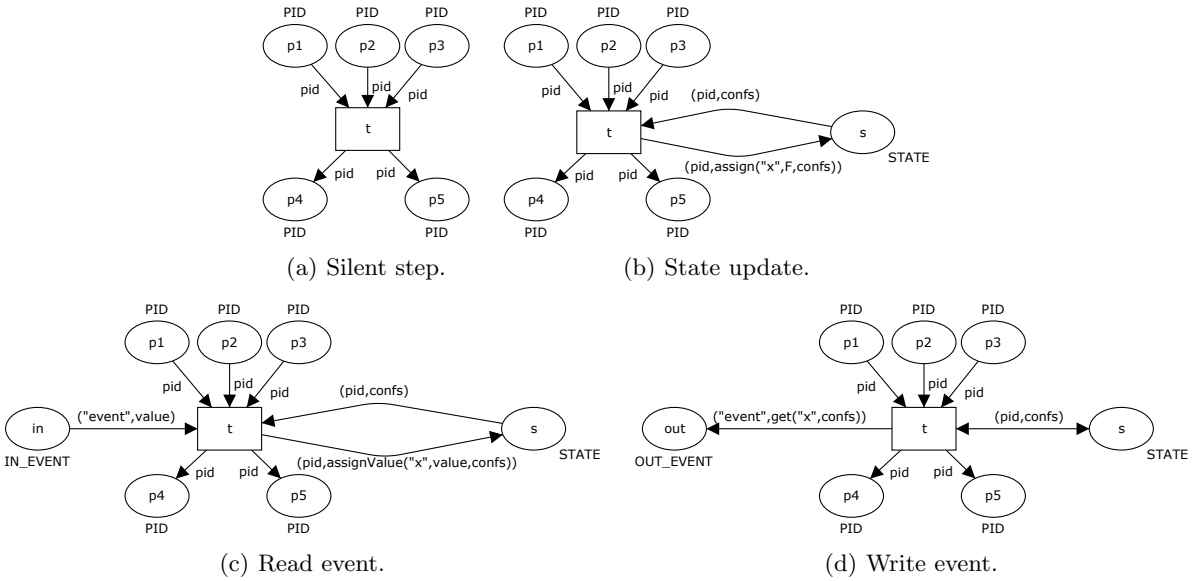
In the following, we will explain each part in the definition. (i) It is possible to use four different kinds of color sets in a CCFN and these are as in Definition 1.

(ii) We say that there are five sets of places. Notice that there are only one start, one end and one state place. The single start and end places are to describe that the control-flow starts in a single point, and ends in a single point. The single state place describe that each CCFN has a global state that is stored on the place p_{state}. A CCFN must have at least one PID place to route the control-flow and any number of in and out event places.

(iii) There are two kinds of transition: Those that we use to initiate and end the process, and a set of transition $T_{regular}$ which we will refer to as regular transitions.

(iv) The arcs consists of two parts where the first is used to setup the CCFN process so the start place is linked to the start transition and so on. Then we state that it is not possible to have two arcs between the same pair of nodes to simplify the function N; this means the function N is now injective and especially that N^{-1} is always uniquely defined. Next we require that it is not possible to make arcs from regular transitions to one of the place p_{start} and transition t_{start} and it is not possible to have an arc going away from the transition t_{end}, except an arc to p_{end}. Next we say that if there is an arc going from p_{state} to a transition t there must be an arc going back from the same t. And, finally if there is an arc from IN_EVENT place to a transition t there may not be an arc to a OUT_EVENT and vice versa. Figure 4 gives four examples of how regular transition may be used in a CCFN, and we will use the terms *silent step,* state update, *read event, and* write event to refer to each of these. Definition 4 formalizes these four possibilities.

(v) As we explained after Definition 1 the places are given place types corresponding to how we intend them to be used.



(a) Silent step. (b) State update.

(c) Read event. (d) Write event.

Fig. 4. Examples of how a regular transition can be used in a CCFN model.

(vi) Transitions that are part of a free choice may have a boolean expression as guard; others have the guard true.

(vii) Figure 4 shows examples of the different cases for the arc expression function. Definition 2 explains the intention of these.

(viii) We initialize the CCFN by adding some multiset of PID on the place p_{start}.

Definition 4 (Transitions in CCFNs). *Let be* $CCFN = (\Sigma, P, T, A, N, C, G, E, I)$ *be a Colored Control Flow Net as in Definition 3. Definition 3 states that there are four distinct sets of transitions that we name as follows:*

Silent step: $\{t \in T | \forall (p_{in}, t), (t, p_{out}) \in N(A) : C(p_{in}) = C(p_{out}) = PID\}$.
State update: $\{t \in T | \exists (t, p_{state}), (p_{state}, t) \in N(A),$
$\forall (p_{in}, t), (t, p_{out}) \in N(A) \setminus \{(t, p_{state}), (p_{state}, t)\} : C(p_{in}) = C(p_{out}) = PID\}$.
Read event: $\{t \in T | \exists (t, p_{state}), (p_{state}, t), (p_{in\_event}, t) \in N(A) : C(p_{in\_event}) = IN\_EVENT,$
$\forall (p_{in}, t), (t, p_{out}) \in N(A) \setminus \{(t, p_{state}), (p_{state}, t), (p_{in\_event}, t)\} : C(p_{in}) = C(p_{out}) = PID\}$.
Write event: $\{t \in T | \exists (t, p_{state}), (p_{state}, t), (t, p_{out\_event}) \in N(A) : C(p_{out\_event}) = OUT\_EVENT,$
$\forall (p_{in}, t), (t, p_{out}) \in N(A) \setminus \{(t, p_{state}), (p_{state}, t), (p_{out\_event}, t)\} : C(p_{in}) = C(p_{out}) = PID\}$.

A CCFN where all places of type STATE, IN_EVENT, and OUT_EVENT are removed should correspond to a Sound Workflow Net (sound WF-net) as defined in [2]; we refer to this version of CCFN as CCFN_{PID}. Although WF-nets have been defined for classical Petri nets it is easy to generalize the definition to CPN as discussed in [3, 4]. The basic requirement is that there is one source place and one sink place and all other nodes (places and transitions) are on a path from source to sink. We can test soundness by making a short-circuited net $\overline{CCFN_{\mathsf{PID}}}$, by adding a new transition t and add two arcs from the end place and to t, and from t to the start place, and finally settings the arc expression of the two arcs to pid. If all transitions in $\overline{CCFN_{\mathsf{PID}}}$ are live and if all places are bounded then we say that the CCFN is sound. Moreover, if all places have multi-set upper bounds $^{++}\sum_{pid \in PID} pid$ we say that the CCFN is safe. In the translation of CCFN, we will assume that the input CCFN is safe and sound in the sense described here.

3 Annotated Java Workflow Nets

In our translation we will need to transform CCFNs to an intermediate form. This form is expressed in a special kind of Petri nets that we call Annotated Java Workflow Nets (AJWNs); see Definition 5. Note that the framework for translating annotated WF-nets was introduced in [7], we refer to Section 6 on a discussion on how our approach related to what was done in [7].

Definition 5 (Annotated Java Workflow Nets). *An Annotated Java Workflow Net is a tuple* $AJWN = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$*, where*

- P *is the set of places.*
- T *is the set of transitions.*
- $F = (P \times T) \cup (T \times P)$ *is a flow relation.*
- $\tau : T \rightarrow JS$*, where* JS *is the set of Java statements.*
- $\tau_G : T \rightarrow BE$*, where* BE *is a boolean expression.*
- $\Gamma = \{(id_1, type_1, value_1), \ldots, (id_n, type_n, value_n)\}$ *is an initial configuration.*
- $\Pi = (\{channel_1^{in}, \ldots, channel_n^{in}\}, \{channel_1^{out}, \ldots, channel_m^{out}\})$ *a pair of in and out channels that carry events.*
- *A single input place* p_s *exists s.t.* $\bullet p_s = \emptyset$ *- i.e. a transition with no input transitions.*
- *A single output place* p_e *exists s.t.* $p_e \bullet = \emptyset$ *- i.e. a transition with no output transitions.*
- $\forall x \in P \cup T : (x, p_{end}) \in F^*$ *(F^* is the transitive closure of F).*
- *The Workflow net (P,T,F) is safe and sound [1].*

In Figure 5 we see a small example of an AJWN. As we will see later the example is actually a translation that we define later of the CCFN of Figure 3.

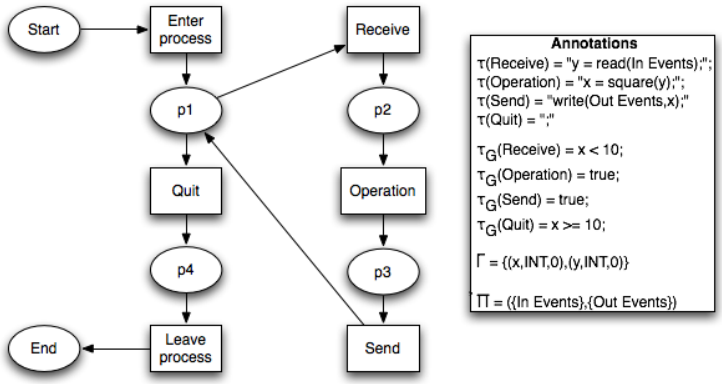


Fig. 5. AJWN of the CCFN in Figure 3.

4 Translation of Colored Control Flow Nets to Annotated Java Workflow Nets

In our approach we map CCFNs to AJFNs, and map the AJFNs to Java. In Definition 6 we give a translation of the first step from CCFNs to AJFNs. An explanation is given after the definition. For an application of the definition we refer to Figure 5, where the AJWN there is a translation of the CCFN in Figure 3.

Definition 6 (Translation of CCFNs to AJFNs). *Let $CCFN = (\Sigma, P^C, T^C, A, N, C, G, E, I)$ be a safe and sound Colored Control Flow Net. We define the corresponding Annotated Java Flow Net $ACFN = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$ as follows:*

(i) *P (places): $P := \{p \in P^C | C(p) = PID\}$.*
(ii) *T (transitions): $T := T^C$.*
(iii) *F (flow relation): $F := \{(p,t),(t',p') \in N(A) | C(p) = C(p') = PID \wedge t, t' \in T^C\}$.*
(iv) *τ (Java annotations): See Definition 4 for a definition of the four possible ways a transition can be used in a CCFN.*

$$\tau(t) = \begin{cases} ";" & \textbf{if } \textit{silent step} \\ "x = F(x_1, \ldots, x_n);" & \textbf{if } \textit{state update} \wedge E(N^{-1}(t, p_{state})) = (pid, assign("x", F(x_1, \ldots, x_n), confs)) \\ "F(x_1, \ldots, x_n);" & \textbf{if } \textit{state update} \wedge E(N^{-1}(t, p_{state})) = (pid, assign("", F(x_1, \ldots, x_n), confs)) \\ "x = value;" & \textbf{if } \textit{state update} \wedge E(N^{-1}(t, p_{state})) = (pid, assignValue("x", value, confs)) \\ "x = read("e", p_{in});" & \textbf{if } \textit{read event} \wedge E(N^{-1}(p_{in}, t)) = ("e", value)) \\ "write("e", x, p_{out});" & \textbf{if } \textit{write event} \wedge E(N^{-1}(t, p_{out})) = ("e", get("x", confs)) \\ "write("e", value, p_{out});" & \textbf{if } \textit{write event} \wedge E(N^{-1}(t, p_{out})) = ("e", value) \end{cases}$$

(v) *τ_G (Guards):*

$$\tau_G(t) = \begin{cases} x\ R\ [VALUE],\ R \textit{ is a boolean operator} & \textbf{if } G(t) = get("x", confs)\ R\ [VALUE] \\ true & otherwise \end{cases}$$

(vi) *Γ (Initial environment): The initial configuration in a CCFN model is specified on the arc (t_{start}, p_{state}) and it is on the form $[(id_1, value_1), \ldots, (id_n, value_n)]$ (see Definition 3). For each pair (id,value) in the initial environment of CCFN, set $\Gamma(id) = value$.*
(vii) *Π (Streams): $\Pi = (\{java\_in\_stream(p) | p \in P_{in\_events}\}, \{java\_out\_stream(p) | p \in P_{out\_events}\})$, where java_in_stream and java_out_stream are streams that implement the* InputStream *and* OutputStream *interfaces in Java.*

The translation is rather straightforward if we look at places and transitions in steps (i) and (ii). In (iii) we map the arcs of the CCFN, where we map all arcs that are not connected to the place p_{state} or any of the in and out event places. (iv) We generate the Java annotations for each transition, by looking at what type of transition it is, and then by looking at the sort of arc expressions used on arcs around it: *silent step* transitions correspond to doing nothing and it is simply translated into the empty statement ";"; *state update* transitions read the current state from the state place and update a value in the current state either by a constant or by calling a function, therefore the statement is either "$x = F(x_1, \ldots, x_n);$" , for some variable x and function F with input $x_1, \ldots, x_n$, or the statement "$x = C;$" for some variable x and constant C; *read event* and *write event* transitions are handled in a similar fashion as the two preceding types. (v) There are two allowed guard expressions in a CCFN: true, and expression on the form get([IDENTIFIER],confs) R value (examples of such an expression is get("x",confs)). (vi) and (vii) simply tell us how to find the initial configuration and which streams are defined. In (vii) we assume that some stream types are defined in the Java code already, so we do not assume they are of type java.io.FileInputStream or other specific types of streams. All we assume is that the input streams are blocking - i.e. a reading call would block if no data is available for reading through the stream and will continue to block until data becomes available.

The time complexity of the translation in Definition 6 is $O(|P \cup T|)$. It simply does a linear sweep of the nodes in the CCFN in mapping to AJWN.

5 Translation of Annotated Java Workflow Nets to Readable Java

Before we describe the translation let us first introduce some definitions. Definition 7 defines how we understand the union of two functions and when a union is possible. This definition is important for the definition of projection in Definition 8.

Definition 7 (Union of two functions). *Let* $F : A_1 \to B, G : A_2 \to B, A_1, A_2 \subseteq A$ *and require that* $\forall x \in A_1 \cap A_2 : F(x) = G(x)$*. Then we define the union of* F *and* G *as*

$$(F \cup G)(x) = \begin{cases} F(x), x \in A_1 \setminus A_2; \\ G(x), \textit{otherwise.} \end{cases}, \forall x \in dom(F) \cup dom(G)$$

Definition 8 states what we mean by a projection, i.e. a restriction, of a AJWN. It simply defines a projection as the subnet you get by selecting a set of nodes, and then include all the arcs that have source and target in that set. The annotation functions of the AJWN are restricted in a similar fashion.

Definition 8 (Projection). *Let* $AJWN = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$ *and* $AJWN' = (P', T', F', \tau', \tau'_G, \Gamma', \Pi')$ *be Annotated Java Workflow Nets and* $X \subseteq P \cup T$ *a set of nodes.* $AJWN|_X = (P \cap X, T \cap X, F \cap (X \times X), \tau|_X, \tau_G|_X, \Gamma|_X, \Pi|_X)$ *is the projection of* $AJWN$ *onto* X*.* $AJWN \cup AJWN' = (P \cup P', T \cup T', F \cup F', \tau \cup \tau', \tau_G \cup \tau_{G'}, \Gamma \cup \Gamma', \Pi \cup \Pi')$ *is the union of* $AJWN$ *and* $AJWN'$*.*

A component as in Definition 9 is a special kind of projection. It is a set of nodes in a AJWN that contain a source and a sink and where all nodes in the component are on a path between source and sink in the AJWN.

Definition 9 (Component). *Let* $AJWN = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$ *be an Annotated Java Workflow Net.* C *is a component of* $AJWN$ *if and only if*

(i) $C \subseteq P \cup T$,
(ii) there exist different source and sink nodes $i_C, o_C \in C$ *such that*
- $\bullet(C \setminus \{i_C\}) \subseteq C \setminus \{o_C\}$,
- $(C \setminus \{o_C\})\bullet \subseteq C \setminus \{i_C\}$, *and*
- $(o_C, i_C) \notin F$.

Definition 10 (Component notations). *Let* $AJWN = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$ *be an Annotated Java Workflow Net and let* C *be a component of* $AJWN$ *with source* i_C *and sink* o_C*. We introduce the following notations and terminology:*

- C *is a PP-component if* $i_C \in P$ *and* $o_C \in P$,
- C *is a TT-component if* $i_C \in T$ *and* $o_C \in T$,
- C *is a PT-component if* $i_C \in P$ *and* $o_C \in T$,
- C *is a TP-component if* $i_C \in T$ *and* $o_C \in P$,
- $\overline{C} = C \setminus \{i_C, o_C\}$,
- $PN||_C =$
  - $PN|_C$ *if* $i_C \in P$ *and* $o_C \in P$,
  - $PN|_C \cup (\{p_{(i,C)}\}, \{i_C\}, \{p_{(i,C)}, i_C\}) \cup (\{p_{(o,C)}\}, \{o_C\}, \{o_C, p_{(o,C)}\})$ *if* $i_C \in T$ *and* $o_C \in T$,[1]
  - $PN|_C \cup (\{p_{(o,C)}\}, \{o_C\}, \{o_C, p_{(o,C)}\})$ *if* $i_C \in P$ *and* $o_C \in T$,
  - $PN|_C \cup (\{p_{(i,C)}\}, \{i_C\}, \{p_{(i,C)}, i_C\})$ *if* $i_C \in T$ *and* $o_C \in P$.
- $[PN]$ *is the set of non-trivial components of* PN*, i.e., all components containing two or more transitions.*

Definition 10 introduces some terminology that is useful in the context of components, such as the projection $PN||_C$. This projection is similar to $PN|_C$ in Definition 8, where a component is padded with places if it is a TT-, PT-, or a TP-component, so that the new projection always yields a AJWN that is bordered by places. It also introduces the concept of non-trivial component denoted $[AJWN]$ for some $AJWN$.

Definition 11 shows how a component may be removed from an AJWN and replaced by a single transition. In [7], we show that a similar kind of folding yields a sound and safe AJWN if the original AJWN was sound and safe.

[1] Note that $p_{(i,C)}$ and $p_{(o,C)}$ are the (fresh) identifiers of the places added to make a transition bordered component place-bordered.

Definition 11 (Fold). *Let $AJWN = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$ be an Annotated Java Workflow Net and let C be a non trivial component of $AJWN$ (i.e., $C \in [AJWN]$). Function* **fold** *replaces C in $AJWN$ by a single transition t_C, i.e., $fold(AJWN, C) = (P', T', F', \tau', \tau'_G, \Gamma', \Pi')$ with:*

- $P' = P \setminus \overline{C}$,
- $T' = (T \setminus C) \cup \{t_C\}$,
- $F' = (F \cap ((P' \times T') \cup (T' \times P'))) \cup \{(p, t_C) | p \in P \cap (\{i_C\} \cup \bullet i_C)\} \cup \{(t_C, p) | p \in P \cap (\{o_C\} \cup o_C \bullet)\}$.
- $\tau' = \tau[t_C := \textbf{translate}(C)]$ (***translate*** *generates Java code from C; for more information on how* ***translate*** *works see Section 5.1).*
- $\tau'_G = \tau_G$.
- $\Gamma' = \Gamma$
- $\Pi = \Pi$

We are now ready to define the algorithm by which we will translate AJWFs. We will explain the steps of the algorithm in detail after the definition. The overall idea behind the algorithm is to take a safe and sound AJWN and reduce it by matching a component, replacing the component by a transition using **fold** that also extends the annotation function by the translation of the component. The predefined components referred to in the algorithm are defined in the next section along with a description of how they are defined.

Definition 12 (Algorithm). *Let $AJWN = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$ be a safe and sound Annotated Java Workflow Net.*

(i) $X := AJWN$
(ii) while $[X] \neq \emptyset$ *(i.e., X contains a non-trivial component)*[2]
(iii) If there is a predefined component $C \in [X]$, select it and goto (vi).
(iv) If there is a component $C \in [X]$ that appears in the component library, select it and goto (vi).
(v) Select a component $C \in [X]$ to be manually mapped into Java and add it to the component library.
(vi) $X := \textbf{fold}(AJWN, C)$ *and return to (ii).*
(vii) Output the Java code attached to the transition in X.

In the next section, we will introduce the predefined component types and these are: SEQUENCE, CHOICE, WHILE, and PARALLEL. The sequence by which the component types are listed is also the precedence that the algorithm use when selecting a predefined algorithm. If the algorithm cannot find any SEQUENCE-component it will look for CHOICE-component, and so on.

If we do not match one of these general predefined components we look in a component library, which is basically a collection of pairs consisting of one AJWN and one matching Java translation. If we find a component in the AJWN that matches the AJWN part of an existing library component, we translate the component based on the translation information found in the library - i.e. we use the Java translation of the component as the annotation for the transition used to replace the component in the AJWN. If no library component can be found, after testing predefined component, the algorithm terminates, since no component in the net could be reduced.

This algorithm can be extended by adding or removing types of predefined and library components, and changing the priority by which the algorithm selects components. For example, the algorithm could be extended to test for PARALLEL-components before SEQUENCE-components.

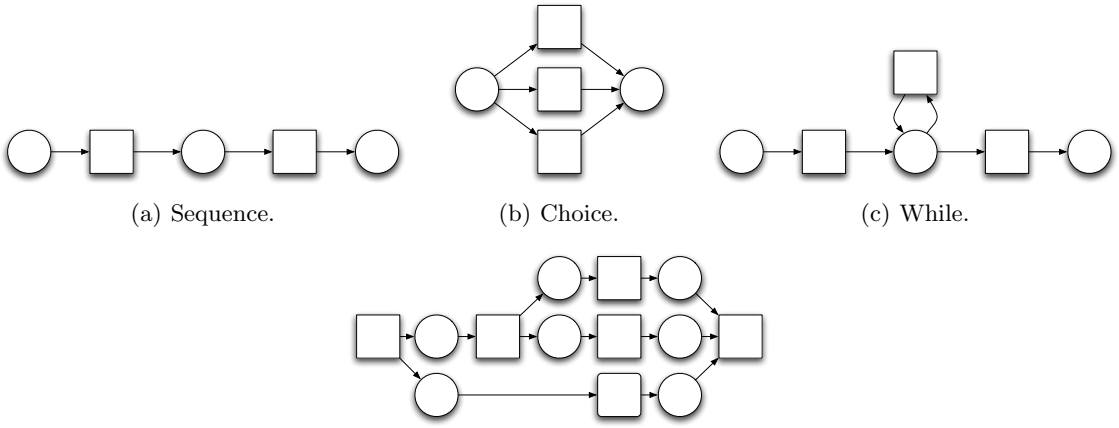
The time complexity of the algorithm in Definition 12 is $O(|T| \cdot 2^{|P \cup T|})$, because the matching of library components may need to compare any combination of nodes in the library component with the nodes input AJWN; $O(\cdot 2^{|P \cup T|})$. This is the most complex operation. We may need to do the operation in each iteration of the loop, which in worst case may need to run $O(|T|)$ times if we match a component in each iteration. This may seem discouraging at first, but in [16] we did a practical investigation on matching library components for real-world WF-nets, and we found that even with a library that had more than 100 components, processing and matching all components took under one second. This was also partly due to an optimized version of subgraph isomorphism for WF-nets that also would apply in this scenario.

[2] Note that this is the case as long as X is not reduced to a WF-net with just a single transition.

5.1 Component Types and Their Translations

This subsection is devoted to describing component we have chosen to translate the function **translate** that was used in the Definition 11 of **fold**. We have chosen some component types that many people know, but as we discussed earlier the algorithm for translating AJWN to Java is extensible by which component translate, so it is possible to add even more components then the ones we present here.

In Figure 6 we show examples of the different component types that we translate. Later on in this section we give formal definitions of these component types and present a translation of them.



(a) Sequence. (b) Choice. (c) While.

(d) Parallel.

Fig. 6. Different component types.

Definition 13 (MAXIMAL SEQUENCE-component). *Let AJWN be a safe and sound Annotated Java Workflow Net, C be a component of AJWN and let* $AJWN|_C = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$.

- *C is a SEQUENCE-component iff* $\forall x \in P \cup T : |\bullet x| \leq 1 \wedge |x \bullet| \leq 1$.
- *C is a MAXIMAL SEQUENCE-component is a component that is not contained in any other SEQUENCE-components.*

We translate a SEQUENCE-component in a straightforward manner: we sort the transitions in T in a list $[t_1, \ldots, t_n]$, where t_i is before t_j iff $(t_i, t_j) \in F^*$. Next, we simply translate the sequence into the Java code: $\tau(t_1); \ldots; \tau(t_n);$.

Definition 14 (CHOICE-component). *Let AJWN be a safe and sound Annotated Java Workflow Net, C a component of AJWN, and* $AJWN|_C = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$.

- *C is a CHOICE-component iff* $P = \{p_i, p_o\}$, $T = p_i\bullet = \bullet p_o$, *and* $\forall t \in p_i\bullet : t\bullet = \{p_o\}$.

We translate a CHOICE-component with transitions $t_1, \ldots, t_n$ by translating $AJWN|_C$ to the Java code: **if** $(\tau_G(t_1))$ $\{\tau(t_1);\}$ **else if** $(\tau_G(t_2))$ $\{\tau(t_2);\}$ $\ldots$ **else** $\{\tau(t_n);\}$.

Definition 15 (WHILE-component). *Let AJWN be a safe and sound Annotated Java Workflow Net, C a component of AJWN, and* $AJWN|_C = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$.

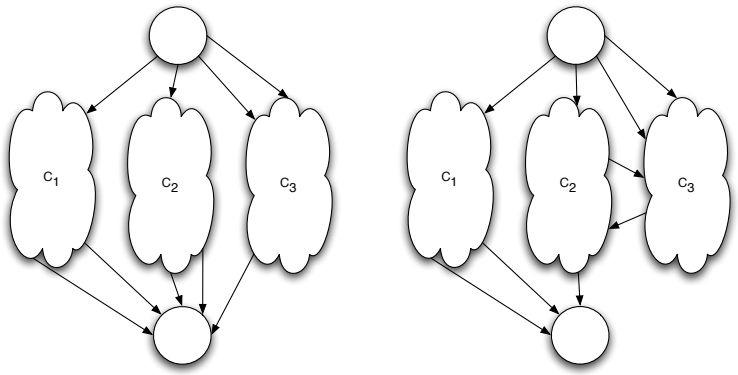
- *C is a WHILE-component iff* $P = \{p\}$, $T = \{t_i, t, t_o\}$ *and* $F = \{(t_i, p), (p, t), (t, p), (p, t_o)\}$.

We translate a WHILE-component with transitions and places given in Definition 15 as the Java code: $\tau(t_i)$; **while**$(\tau_G(t) \wedge \neg\tau_G(t_o))$ $\{\tau(t);\}$ $\tau(t_o)$;.

Definition 16 (MAXIMAL PARALLEL-component). *Let AJWN be a safe and sound Annotated Java Workflow Net, C a component of AJWN, and* $AJWN|_C = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$.

- *C is a PARALLEL-component iff it is an acyclic marked graph; i.e.* $\forall p \in P : |\bullet p| \leq 1 \wedge |p \bullet| \leq 1$ *and* $\forall x \in P \cup T : (x, x) \notin F^*$.
- *C is a MAXIMAL PARALLEL-component is a PARALLEL-component that is not contained in a larger PARALLEL-component.*

To translate a PARALLEL-component we need to introduce the concept of a clean synchronization and branches in Definition 17. For an example of PARALLEL-components with clean and unclean synchronization please refer to Figure 7.



(a) Clean synchronization with three branches.

(b) Unclean synchronization; here we do not say that there are any branches.

Fig. 7. Two examples of a PARALLEL-components.

Definition 17 (Clean Synchronization and Branches). *Let AJWN be a safe and sound Annotated Java Workflow Net, C a component of AJWN, and* $AJWN|_C = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$. *Assume with no loss of generality that C is a TT-component and that the source and sink are* $t_i, t_o \in T$. *If for each* $p_i \in t_i\bullet$ *we can find a* $p_o \in \bullet t_o$, *such that* p_i *and* p_o *are source and sink of a component and* p_i, p_o *are only used in a single component, then we say that the PARALLEL-component has clean synchronization.*

We call the components we find between t_i *and* t_o *branches of* $AJWN|_C$, *and we say that they are disjoint.*

A MAXIMAL PARALLEL-component that only has clean synchronization is easy to translate to Java since it corresponds to a split in the WF-net that is later synchronized and no synchronization happens before this final synchronization. This simply corresponds to starting n Java threads in parallel and waiting for all of them to finish. However, clean synchronization is not always the case in MAXIMAL PARALLEL-component, which is why the following translation become more verbose.

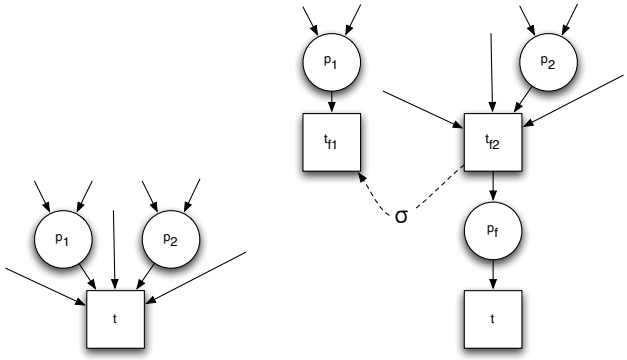
When we translate a MAXIMAL PARALLEL-component, we assume without loss of generality that it has source and sink $t_i, t_o \in T$.

We split the translation of MAXIMAL PARALLEL-component $AJWN|_C$ in two cases: First we look at $AJWN|_C$ with clean synchronization (see Figure 7(a)), and later on we take the other case. In this case $AJWN|_C$ has n branches $C_1, \ldots, C_n$. Here we generate the following Java code:

```
τ(t_i);
Thread t_{C_1} = new Thread() {public void run() {translate(C_1);}}
...;
Thread t_{C_n} = new Thread() {public void run() {translate(C_n);}}
t_{C_1}.start(); ...; t_{C_n}.start();
τ(t_o);
```

In case the $AJWN|_C$ does not have clean synchronization (see Figure 7(b)), we have to do something a little more complicated. Before we begin to explain the steps, let us define two functions $\sigma : T \to T$ and $\kappa : T \to \mathcal{P}(T)$ that we will use to define relations between two transitions, σ, a function $\rho : T \to$ *legal Java identifier* to relate transitions to Java thread identifiers and a function $\varphi : T \to$ *Java code* to relate transitions to some Java code. The five steps to translate $AJWN|_C$ are the following:

1. For each join transition t ($|\bullet t| > 1$) we look at p_1 and p_2, where $\{p_1, p_2\} \in \bullet t$. We introduce fresh place p_f, and fresh transitions t_{f_1}, t_{f_2} and update C by setting $P := P \cup \{p_f\}$, $T := T \cup \{t_{f_1}, t_{f_2}\}$, and $F := (F \cup \{(p_1, t_{f_1}), (p_2, t_{f_2}), (t_{f_2}, p_f), (p_f, t)\}) \setminus \{(p_1, t)\}$. Moreover we set $\sigma(t_{f_2}) = t_{f_1}$. We continue doing this until all join transitions are removed. This step removes all synchronization point and this means that C is converted to a tree where t_i is the root.



(a) Before. (b) After.

Fig. 8. Example of how step 1 in the MAXIMAL PARALLEL-component translation works.

2. For each split transition t, we find all reachable transitions $t_1, \ldots, t_n$. Set $\kappa(t) = \{t_1, \ldots, t_n\}$. This step determines which threads will generated in the Java code.
3. Now we generate identifiers for each thread in the Java code. For each $t \in dom(\kappa)$ generate a fresh identifier id_t and set $\rho(t) = id_t$.
4. In this step we need to map each transition in C to some Java code. For the original members of C we have τ that carry the Java code of those, but for the new transitions that we added in step 1 we need to do something else. For this reason we introduce the extended annotation function τ_{ext} that we define as follows:

$$\tau_{ext}(t) = \begin{cases} \tau(t), & \text{if } t \in C; \\ join(\sigma(t)), & \text{if } t \in dom(\sigma); \\ ; & \text{otherwise.} \end{cases}$$

join(t) is a short hand for synchronizing a perhaps not yet scheduled thread. We can write it as "while (t.getState()==Thread.State.NEW) {try {wait();}catch (InteruptedException ie) {} t.join();".

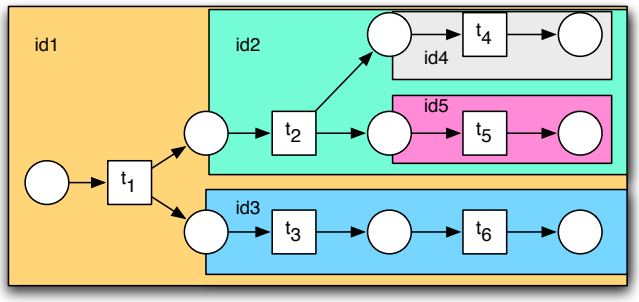


Fig. 9. Illustration of step 2 and 3 in the MAXIMAL PARALLEL-component translation.

Each member $t \in dom(\kappa)$ is a tree that starts with a sequence and either ends in that sequence or in a split by a split transition. In the first case, assume that the sequence of transitions is $t_1, \ldots, t_n$ and set $\varphi(t) = \tau_{ext}(t_1); \ldots; \tau_{ext}(t_n);$. If the tree from t is not just a sequence but contains a split, then the tree will be a sequence with transition $t_1, \ldots, t_n$ followed by a transition split t_s where $\kappa(t_s) = \{t_{s_1}, \ldots, t_{s_n}\}$. In this case we set $\varphi(t) =$ "$\tau_{ext}(t_1); \ldots; \tau_{ext}(t_1); \rho(t_{s_1}).start(); \ldots \rho(t_{s_n}).start();$".

5. At this point assume $dom(\kappa) = \{t_1, \ldots, t_n\}$. We now generate the Java code. The code first declares all threads that we need and then starts the initial thread:

```
1 Thread ρ(t1) = new Thread() {public void run() {φ(t1);}});
2 ...;
3 Thread ρ(tn) = new Thread() {public void run() {φ(tn);}});
4 ρ(ti).start();
```

In step 1 we remove all joins in $AJWN|_C$ and an example of this is found in Figure 8, which shows how a single join transition is transformed. Figure 9 illustrates the steps 2 and 3. In this figure $\kappa(t_1) = \{t_2, t_3, t_4, t_5, t_6\}, \kappa(t_2) = \{t_4, t_5\}, \kappa(t_3) = \{t_6\}, \kappa(t_4) = \kappa(t_5) = \kappa(t_6) = \emptyset$.

This final translation of acyclic marked graphs concludes the algorithm. Next we will show a small example of translating AJWNs to Java.

5.2 Example of the Translation of Annotated Java Workflow Net to Java

In this section we will look at a small generic example of an application of translation algorithm presented in the previous section. The example in Figure 10 focuses on the translation of the structural parts of the AJWN, so we do not consider the annotations of the AJWN since they do not affect the algorithm; we set the annotations of transition tN to $\tau(tN) =$ "$tN();$" and $\tau_G(tN) = gN$, where gN is some boolean expression. In the figure, all components are boxed and named in order to make it easier to see how the algorithm matches components. Our algorithm reduces the net by folding components in the following order:

1. Match MAXIMAL SEQUENCE-component C1 {p4, p5, p9, p13, t3, t8, t10}, replace with transition t_{C1}, and extend τ so that $\tau(t_{C1}) =$ **translate**$(C1)$.
2. Match MAXIMAL SEQUENCE-component C2 {p8, t6, t14}, replace with transition t_{C2}, and extend τ so that $\tau(t_{C2}) =$ **translate**$(C2)$.
3. Match CHOICE-component C3 {p2, p6, t4, t5}, replace with transition t_{C3}, and extend τ so that $\tau(t_{C3}) =$ **translate**$(C3)$.
4. Match WHILE-component C4 {p8, t2, t7, t_{C2}} and replace with transition t_{C4}, and extend τ so that $\tau(t_{C4}) =$ **translate**$(C4)$.

5. Match MAXIMAL PARALLEL-COMPONENT C5 containing the rest of the net, replace with transition t_{C5}, and extend τ so that $\tau(t_{C5}) = \textbf{translate}(C5)$.

We map the AJWN into the Java code in Listing 1.1 and we have marked the start and end of the translation of each component.

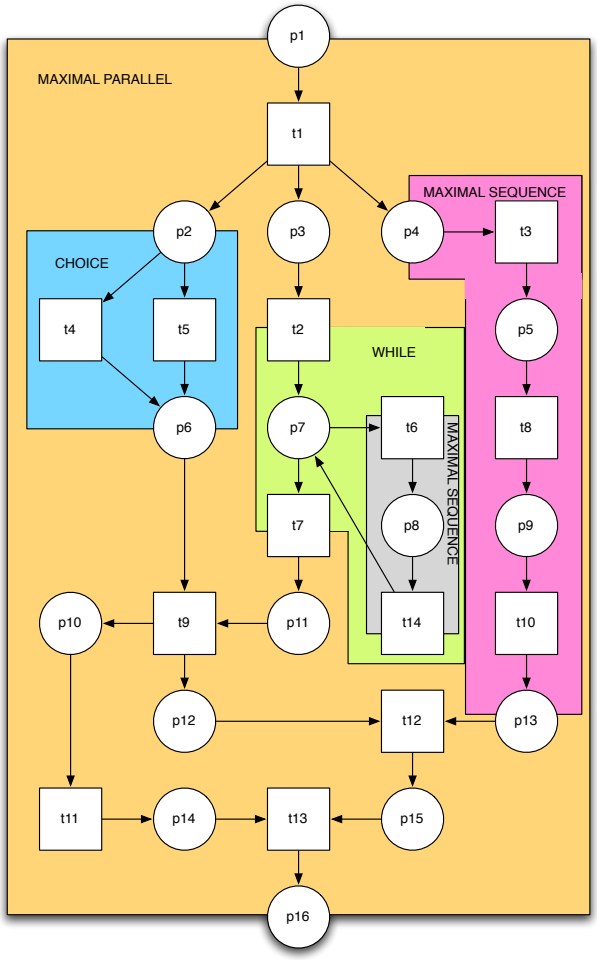


Fig. 10. AJWN to be folded using Definition 12. Notice that all the components of the AJWN are shown along with the type of the component.

6 Related Work

In [20], Wieringa describes his perception of a reactive system. Our approach matches well with this view on reactive systems as a part of the approach to describing the behavior of such systems. CCFN fit in the behavioral descriptions category in a tool box for reactive systems, where Wieringa uses state charts for that purpose. Other tools include E/R-diagrams to model the contents of the messages passed from the system to the environment and vice versa, and use communication diagrams, a variant of data flow charts, to describe

Listing 1.1. Translation of the AJWN in Figure 10

```
// Start: MAXIMAL PARALLEL
final Thread t1 = new Thread(new Runnable()
    {public void run() {
        if (g1) {t4();} // Start: CHOICE
        else {t5();} // End: CHOICE
    }});
final Thread t2 = new Thread(new Runnable()
    {public void run() {t11();}});
final Thread t3 = new Thread(new Runnable()
    {public void run() {}});
final Thread t4 = new Thread(new Runnable()
    {public void run() {
        t2(); // Start: WHILE
        while(g2) {
            t6(); // Start: MAXIMAL SEQUENCE
            t14(); // End: MAXIMAL SEQUENCE
        }
        t7(); // End: WHILE
        while (t1.getState() == Thread.State.NEW)
            try{wait();}
            catch (InteruptedException ie) {}
        t1.join();
        t9();
        t3.start();
    }});
final Thread t5 = new Thread(new Runnable()
    {public void run() {
        t3(); // Start: MAXIMAL SEQUENCE
        t8();
        t10(); // End: MAXIMAL SEQUENCE
        while (t4.getState() == Thread.State.NEW)
            try{wait();}
            catch (InteruptedException ie) {}
        t4.join();
        t12();
    }});
t1();
t1.start(); t4.start(); t5.start();
while (t2.getState() == Thread.State.NEW)
    try{wait();}
    catch (InteruptedException ie) {}
t2.join(); t5.join();
t13(); // End: MAXIMAL PARALLEL
```

the channels, processes, and storage in order to show how the system is structured internally and how it communicates with its environment.

It is fair to compare the class of CCFN to Colored Workflow Nets (CWNs) as defined and used in [6, 15]

in the sense that it was developed to mirror an underlying computational paradigm: CCFN is intended to be used for modeling systems in an imperative manner, whereas CWNs are aimed at developing models representing resources, tasks, and case perspectives in the domain of workflow systems. This was enforced by the definition of CWN. CWNs were used as a starting point for mapping CWNs to annotated WF-nets, and these WF-nets were mapped to BPEL.

In [7], we mapped a special kind of annotated WF-net to BPEL. This is similar to what we did in Section 5 where we focused on a different sort of annotated WF-net and other component types, although some types are the same. The translations from the WF-net to BPEL and from CPN to Java are different, since BPEL is a language that directly supports many of the structural component types, whereas the imperative nature of Java complicates the translation.

Philippi [18] outlines three methods for translating high-level Petri nets (such as CPNs) to imperative code: structural-, simulation-, and reachability-based. He dismisses a structural approach as we propose in this paper, since he feels that high-level Petri net cover more behavioral constructs than what common imperative languages such as Java provide, so he thinks that such a translation is not possible. Instead he proposes a simulation-based approach which means that he translates the high-level Petri nets to an interpreter that interprets the state of the high-level Petri net. In other words, he constructs an interpreter that describe how transitions fire by calculating binding elements, what tokens are consumed and generated, and other Petri net related parts. This means that the code he generates is difficult to read since e.g. a sequence of transitions are not mapped to a sequence of statements, but instead a high-level CPN interpreter.

Mortensen [17] translates CPNs to executable code. His method is a simulation-based one in which he takes the simulation image of a running CPN and extract the code it was build from (CPNs in his paper are compiled to Standard ML code), and maps this code to some implementation platform, including Java. He is also able to simply execute the simulation image in a Standard ML environment. In contrast to Philipi, Mortensen's approach is simply technical since he does not provide a generation process for the interpreter, but simply extracts it from already generated code from Design/CPN. In the paper, he mentions that he does not have any experience with the compilation to Java, but expects that it will be very slow. By his approach, Mortensen will not have any control over how the generated Java code will look like and does therefore not have any possibility to tweak the appearance of the generated code, as we are able to in our approach.

7 Conclusion and Future Work

The translation of this paper is intended as a move to bridge the gap between CPNs and Java and we feel that this is a step in the right direction. We think that translating any CPN to Java is not feasible, so we introduced CCFN as a restriction of CPN, to get a subset of CPN that could be mapped into readable Java. Although CCFN is simple in many ways, we think it is powerful enough to express many of the control-flows used in Java programs.

To help the translation to Java we introduced AJWN. Besides making the translation more smooth, we think it make the overall translation more extensible in two ways: (1) In Definition 4 we describe the four ways we allow a transition to occur in a CCFN and we were later able to present translations into Java statements of these ways. If anyone can find more ways to map transitions to Java statements they just have to change the definition of CCFN and the translation of these changes into Java statements, and leave the rest unchanged. (2) On the other hand, if someone has a better translations for component types or other component types, they can simply change the second part of the translation, and leave the first part unchanged. The algorithm in this paper is described in such a way that it is possible to implement this in a compiler.

The time complexity for the translation equals the sum two measures: (1) the complexity of the translation from CCFN to AJWN, which is $O(|P \cup T|)$ (P and T places and transition in CCFN), and (2) the complexity of the translation from AJWN to Java, which is $O(|T| \cdot 2^{|P \cup T|})$. In total, the complexity is $O(|T| \cdot 2^{|P \cup T|})$. However, as discussed in the end of Section 5, this is not an issue, since it is possible to find algorithms

that, although they do not change the time complexity, they are so effective that translating CCFNs is not a problem in practice.

In this paper, we have introduced an algorithm for transforming CCFNs to Java code. There are, however, many things that would be sensible extension to the algorithm, but also areas where it would be interesting to test the applicability of the approach. In this section we will discuss both of these issues.

Extensions to Translation: Hierarchy The modeler should be able to modularize CCFN to keep models manageable. To do this, Definition 3 needs to be changed to allow for hierarchical constructs. Obviously, this will change the way we need to map the CCFN. Mapping could be handled in one of two ways: (1) mapping flattens the CCFN process definition into one net with no substitution transitions, and uses the translation algorithm that we have presented in this paper, or; (2) each page in the CCFN hierarchy is separately mapped to a Java class. The last proposal is the most complicated one since we have to give a Java translation of how and when control is transfered from a super page to a subpage and vice versa.

Extensions to Translation: Better Reactive System Support CCFNs were partly designed to be easy to map to Java, but also to support certain elements in the reactive systems terminology as described by Wieringa [20]. We have places for input and output events that in the sense of Wieringa model named events. It would be desirable to enable the modeling of interaction through shared states existing in the interface between the system and the environment. Such a shared state would be readable by both parties while only one party would be able to change (/control) the state. For example, a state representing a thermometer can be altered by the environment and read by the system. This extension can be done by extending Definition 3 in order to allow a CCFN to contain such places. The type of the place could simply be a CONFIGURATION_LIST.

Extensions to Translation: State machines Since a program written in an imperative language without parallelism is best described as a state machine, it would be desirable to be able to map components of AJWNs that are state machines - i.e. to be able to recognize components with the structural properties of state machines. [8, 12] describe algorithms to compile a goto graph into code consisting of loops and alternations. A state machine in a Petri net can be viewed as a goto graph, so it should be possible to adapt their theory to handle state machines in AJWNs as well.

Uses of the Translation: Model-driven Software Development It would be interesting to use this translation approach in a model-driven software development projects such as those presented in [6,15]. The goal of the system should then be some reactive system.

Uses of the Translation: Library Components In the algorithm presented in Definition 12 we allow the user to provide library components if none of the standard components could be matched. This introduces an area we have not touched much in this paper: Not all AJWNs may be reducible. In fact, in [16] 100 models were reduced using the same component definition and folding technique as the one presented in this paper, and 76 unique non-reducible components were found. The models were designed by graduate students and they were encouraged to implement advanced workflow patterns [5]. Although, the models were generally more complicated than what could be expected, it still shows that using this method for translating a graph, you must anticipate that some manual work is needed. It would therefore be interesting to study which non-reducible components that are typically found in the area of reactive system design.

References

1. W.M.P. van der Aalst. Verification of Workflow Nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. Springer-Verlag, Berlin, 1997.
2. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

3. W.M.P. van der Aalst. Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 1–65. Springer-Verlag, Berlin, 2004.
4. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2004.
5. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
6. W.M.P. van der Aalst, J.B. Jørgensen, and K.B. Lassen. Let's Go All the Way: From Requirements via Colored Workflow Nets to a BPEL Implementation of a New Bank System Paper. In R. Meersman and Z. Tari et al., editors, *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005*, volume 3760 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, Berlin, 2005.
7. W.M.P. van der Aalst and K.B. Lassen. Translating Unstructured Workflow Processes to Readable BPEL: Theory and Implementation. *Information and Software Technology*, 2006.
8. Zahira Ammarguellat. A Control-Flow Normalization Algorithm and its Complexity. *IEEE Trans. Softw. Eng.*, 18(3):237–251, 1992.
9. CPN Tools. www.daimi.au.dk/CPNTools.
10. João Miguel Fernandes, Jens Bæk Jørgensen, and Simon Tjell. Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net. In *Proc. of 6th International Workshop on Scenarios and State Machines (SCESM) at ICSE 2007*, 2007.
11. João Miguel Fernandes, Simon Tjell, and Jens Bæk Jørgensen. Requirements Engineering for Reactive Systems: Coloured Petri Nets for an Elevator Controller. In *Proc. of 14th Asia-Pacific Software Engineering Conference (APSEC)*, 2007.
12. Rainer Hauser and Jana Koehler. Compiling process graphs into executable code. In Gabor Karsai and Eelco Visser, editors, *GPCE*, volume 3286 of *Lecture Notes in Computer Science*, pages 317–336. Springer, 2004.
13. K. Jensen. *Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1992.
14. Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 2007.
15. Jens Bæk Jørgensen, Kristian Bisgaard Lassen, and Wil M.P. van der Aalst. From Task Descriptions via Coloured Petri Nets Towards an Implementation of a New Electronic Patient Record. *Software Tools for Technology Transfer*, 2007.
16. Kristian Bisgaard Lassen and Wil M.P. van der Aalst. WorkflowNet2BPEL4WS: A Tool for Translating Unstructured Workflow Processes to Readable BPEL. In R. Meersman and Z.Tari, editors, *CoopIS/DOA/ODBASE*, 2006.
17. K. H. Mortensen. Automatic code generation from coloured petri nets for an access control system. In *Kurt Jensen (ed.): Second Workshop on Practical Use of Coloured Petri Nets and Design/CPN, Aarhus, Denmark*, pages 41–58, October 1999. InternalNote: Submitted by: khm@daimi.au.dk.
18. Stephan Philippi. Automatic code generation from high-level petri-nets for model driven systems engineering. *Journal of Systems and Software*, 79(10):1444–1455, 2006.
19. Simon Tjell. Distinguishing Environment and System in Coloured Petri Net Models of Reactive Systems. In *IEEE Second International Symposium on Industrial Embedded Systems*, 2007.
20. R. J. Wieringa. *Design Methods for Software Systems: YOURDON, Statemate and UML*. Science & Technology Books, 2002.