A Programming Language Type Checking Approach to Workflow Net Verification

Genio Brylle Viernes

Department of Computer Science, University of the Philippines Diliman, Diliman Quezon City, Philippines
gcviernes@up.edu.ph

Abstract. A workflow net is a computational tool for modeling business processes. Their verification is crucial for ensuring that the processes they model follow a criteria of correctness. In this paper, we observe that workflow net constructs can be mapped to programming language features. This mapping allows the encoding of a workflow net to a simple language with a static semantics based on workflow net semantics. We then make a custom type system for checking this semantics, effectively checking the correctness of the encoded workflow net.

Keywords: Workflow Nets, Programming Languages, Type Systems

1 Preliminaries

1.1 Workflow nets

Workflow nets are a computational model for modeling and analyzing business processes, or simply, workflows. They are a particular class of a much general computational model called Petri nets.

Definition 1. *A Petri net is a triple (P, T, F), where P is a finite set of places, T is a finite set of transitions, with $P \cap T = \varnothing$, and $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (the flow relation from places to transitions and transitions to places) [1].*

At any time, a place can have zero or more *tokens*. A transition is said to be *enabled* if all its *input places* (those places that connect to it via some arc) have at least one token in them each. An enabled transition may fire, consuming one token from each of its input places, and produces one token to each of its *output places* (those places that the transition points to via some arc) [1].

We narrow down the scope of this paper to a class of Petri nets called workfow nets, that have only a single input place *start*, only a single output place *end*, and no dead-end paths, that is, all places/transitions are on some path from the start to the end place. These unique start and unique end places model processes of the kind where there is a definite beginning and a definite finish.

© The Author(s) 2026
J. Caro et al. (eds.), *Proceedings of the Workshop on Computation: Theory and Practice (WCTP 2025)*, Atlantis Highlights in Computer Sciences 24,
https://doi.org/10.2991/978-94-6239-638-8_7

Definition 2. *A workflow net (WF net) is a Petri net where there is an* $s \in P$ *that has no incoming arcs, i.e., a unique start place, an* $e \in P$ *that has no outgoing arcs, i.e., a unique end place, and every node* $n \in P \cup T$ *is on some path from* s *to* e*, that is, if a short-circuiting arc is used to connect* e *to* s*, the resulting directed graph is strongly connected, i.e., there are no dead ends.*

Figure 1 shows a Petri net that is not a workflow net because of a loose end.

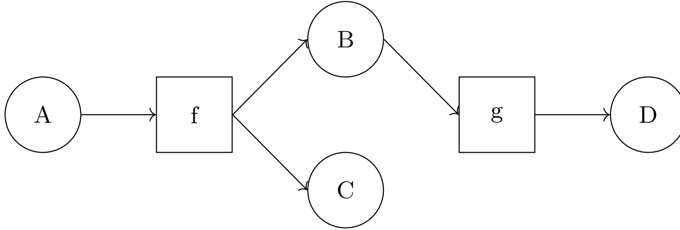


Fig. 1. A Petri net that is NOT a workflow net because of a loose end at node C.

Figure 2 shows Figure 1 modified with a short-circuiting arc to test for strong connectedness, which fails.

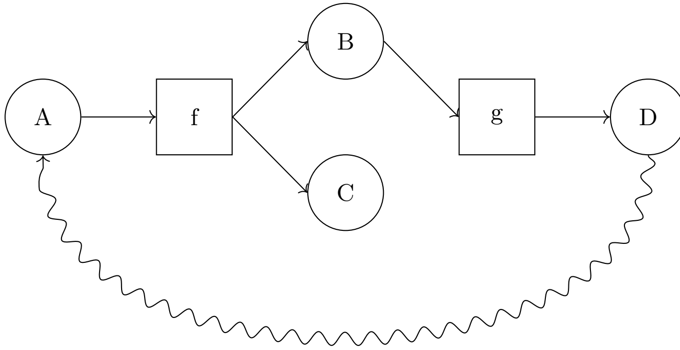


Fig. 2. The short-circuiting arc represented by the squiggly arrow going from end node D to start node A shows that the resulting graph is not strongly connected, as there is no path from node C to anywhere else; this concludes that Figure 1 does not qualify as a workflow net.

1.2 Formal Type Systems

A formal type system is a kind of formal proof system: a collection of axioms and rules used to carry out deductions to prove the validity of statements. A type

system in particular is concerned with the validity of typing relations between terms and types in programs.

The description of a type system starts with a collection of sentences called *judgments*. They take on the form $\Gamma \vdash \xi$ where ξ is an assertion and its free variables are declared in Γ, and Γ represents a *static typing environment* (also called *context*), which is a collection of known distinct variables and their types. The form of ξ varies from judgment to judgment. Overall, a judgment is read as "Γ entails ξ", that is, assertion ξ is possible under Γ.

The most important judgment is $\Gamma \vdash M : A$, which says that term M has type A in Γ. A valid example of this is the judgment $\varnothing \vdash true : Boolean$, which says that in the empty environment (the environment is empty because the terms on the right side of $\vdash$ is just one constant, thus has no free variables), a value of *true* is of type *Boolean* (assuming the typical interpretation of *true*). An invalid example is $\Gamma \vdash true : Number$. The validity of judgments formalizes the notion of *well-typing* of terms in programs.

Valid judgments aren't enough to describe a type system on their own. Valid judgments are used to assert the validity of other judgments in form of type rules. Type rules take on the form:

$$\frac{\Gamma_1 \vdash \xi_1 \qquad \Gamma_2 \vdash \xi_2 \qquad ... \qquad \Gamma_n \vdash \xi_n \qquad (Annotations)}{\Gamma \vdash \xi} \text{ (RULE NAME)}$$

Judgments above the horizontal line are called *premises* and the single judgment below is called the *conclusion*. The form is read as an implication: if all premises are true, then the conclusion is true. A type rule must always have exactly one conclusion but it can have zero promises, in which case, it is an axiom of the type system. A type rule can have annotations as well, which are side conditions (not of the judgmental form) that must be met for the conclusion to still hold. A type rule with zero judgmental-form premises and only side conditions are still considered axioms. Type rules are also given names, in the same way inference rules have names in propositional logic.

Type rules are used in a *derivation*, a tree of invoked type rules with axioms at the top and some judgment at the bottom, whose conclusion is the one we're checking the validity of, and that intermediate judgments are obtained by applying the appropriate type rules.

Consider the simple language of natural numbers and addition, and its type system consisting of only two rules. The first rule says that an integer is of type $Natural$:

$$\frac{(n = 0, 1, 2, 3, ...)}{\Gamma \vdash n : Natural} \text{ (VAL } n\text{)}$$

The second rule says that the addition of two integers is of type $Natural$:

$$\frac{\Gamma \vdash M : Natural \qquad \Gamma \vdash N : Natural}{\Gamma \vdash M + N : Natural} \text{ (VAL +)}$$

These two rules formalize the obvious in this small language, but formalizing details such as these is what type systems are made for: checking the validity of a derivation then is nothing but checking if each invoked rule in the tree is correctly applied, guided by the name at every step. The following valid derivation says that the term $1 + 2$ is of type $Natural$; we say that the term $1 + 2$ is *well-typed*:

$$\dfrac{\dfrac{}{\varnothing \vdash 1 : Natural}\ \textsc{by}\ (\textsc{Val}\ n) \qquad \dfrac{}{\varnothing \vdash 2 : Natural}\ \textsc{by}\ (\textsc{Val}\ n)}{\varnothing \vdash 1 + 2 : Natural}\ \textsc{by}\ (\textsc{Val}\ +)$$

If we are to include the *Boolean true* in our language, and add an axiom with the conclusion $\Gamma \vdash true : Boolean$, but forget to add a rule for how this interacts with addition, we get the following derivation that gets 'stuck' for the term $2 + true$, i.e., the term $2 + true$ is not well-typed in this language:

$$\dfrac{}{\varnothing \vdash 2 + true :?}\ (\textsc{stuck, no rule matches})$$

We could, however, add the following rule:

$$\dfrac{\Gamma \vdash M : Natural \qquad \Gamma \vdash N : Boolean}{\Gamma \vdash M + N : Natural}\ (\textsc{Val}\ +')$$

with the intention of interpreting *true* as the integer 1, to make terms that do 'addition of Booleans and Naturals' valid in the language.

A type system can be customized this way to allow terms in a language to admit some non-conventional meanings, but in essence: type systems rule out terms that are not well-typed.

For a more comprehensive introduction to formal type systems, refer to [3], where this section is based from.

2 Workflow nets as a programming language

2.1 Places as variables, transitions as functions

Workflow net components can be modeled as programming language features. Specifically, places can be encoded as variables and transitions can be encoded as functions. These variables are annotated with a type and the type signatures of the functions are based on the kind of transition they are encoding and the types of their input and output (places).

From here on, we think of places as variables and functions as transitions, and vice versa, with respect to properties related to both. For example, when we talk about "the type of a place", we refer to the type of the variable that the place is encoded to. Another example is "the function type of a transition", which refers to the function type signature of an encoded transition. Another is "the start variable", which refers to the variable that encodes the start place of the WF net.

2.2 Encoding WF net transitions as functions

A transition can have one input place and one output place at its most basic form. Its more advanced forms can have input (or output) that is a choice from one or more places, or a collection of two or more places. The notion of "choice" and "collection" of places can be modeled using composite data types, *sum* and *product types*, respectively.

Definition 3. *A sum type (or union type [3]), denoted by $A + B$, refers to the type of some singular value that can either be of type A or type B. Sum types can be generalized to involve more than two types with $A_1 + A_2 + ... + A_n$, which refers the type of some value with a possible type A_i, where $1 <= i <= n$.*

Definition 4. *A product type, denoted by $A \times B$, refers to the type of some value that has two components, the first one having type A, and the second one having type B. They are associated with projection functions that extract components from values. For a value of type $A \times B$, there is a* projection function `first` *that gets the value of type A, and a projection function* `second` *that gets the value of type B [3]. Product types can be generalized to involve more than two types with $A_1 \times A_2 \times ... \times A_n$, which refers to the type of some value with an n number of components, where the ith component has type A_i, for $1 \leq i \leq n$.*

It will be useful later to talk about the non-composite types that constitute a composite data type, for example, in a product type $A \times B \times C$, its constituent types are A, B, and C.

Definition 5. *A constituent type of a product or sum type refers to the non-composite types that constitute the type.*

In Figure 3, we have a mapping of several WF net transitions (derived from [2]) to their respective function type signature equivalents. By convention, we give uppercase letters for places as their name (and type) and lowercase letters for transitions as their (function) name.

Note the discrepancy of the implied (possibly) inclusive choice interpretation of OR-join and OR-split transitions compared to the definition of sum types in Definition 3, which is that of exclusive choice (XOR). We will stick to how these transitions are named in the literature but our interpretation of them will be of XOR for simplicity. As convention, AND-split and OR-split transitions are graphically differentiated by the dashed lines connected to the OR-split, compared to the solid line connected to the AND-split (suppose that the dashed lines indicate exclusive choice from one of the places they connect to). This convention goes for AND-join and OR-join transitions as well.

It's possible that a transition can have two split/join constructs, one per input and output side. Figure 4 shows this.

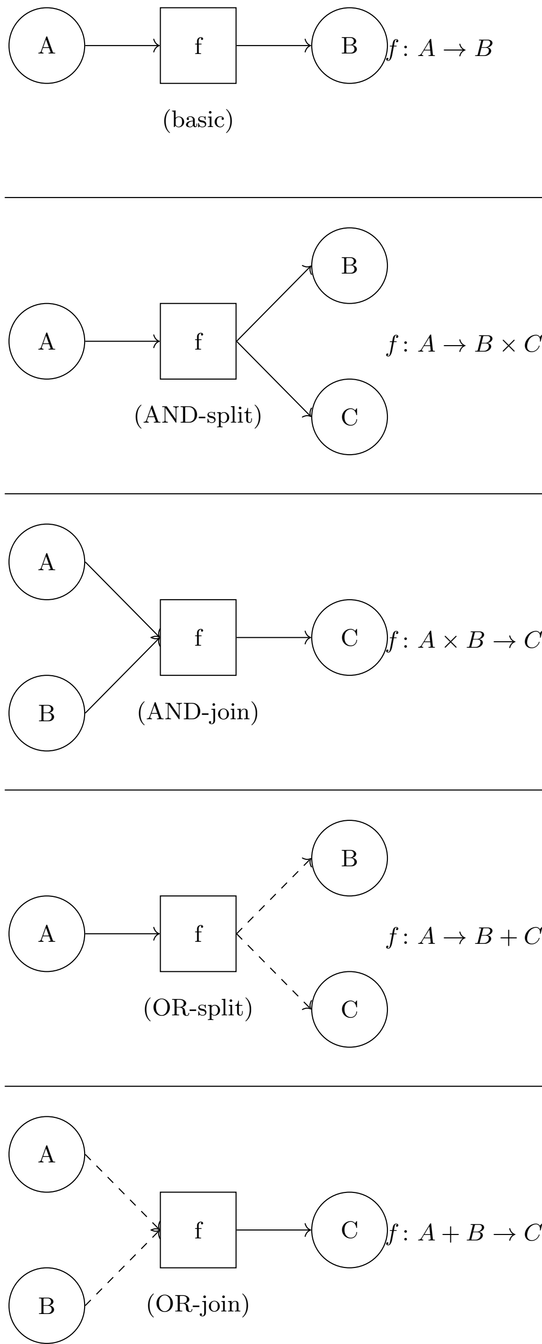


Fig. 3. Transitions and their equivalent type signatures

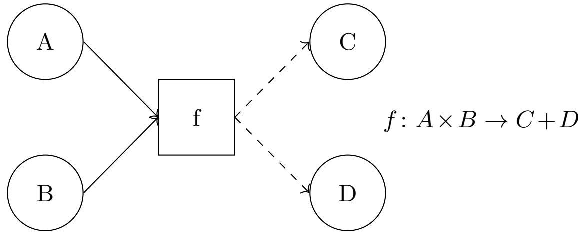


Fig. 4. A transition with an AND-join input and an OR-split output, and its equivalent function type.

2.3 Encoding entire WF nets

This is a simple, straightforward scheme for encoding some input WF net into some programming language.

1. For every place, we associate with it a variable of some type. The type will be represented by an uppercase letter and the name of the variable can simply be the type name but in lowercase. The types are pairwise distinct.
2. For every transition, we associate with it an appropriate function type involving the types of its input and output places based on Figure 3. We give it a name of the form f_j, where $1 <= j <= m$, and m is the number of transitions in the net.

These two steps produce variables and functions and their associated types in isolation. They are not enough; we still need an interpretation of the arrows in the WF net to get a semblance of a programming language.

1. Arrows from transitions to places are interpreted as variable assignments.
2. Arrows from places to transitions are interpreted as function calls.

The purpose of the scheme is the following. With a WF net encoded as variables, functions, assignments and function calls, it can be written into source code of some programming language with compile-time type checking, whose failure of type checking indicates some incorrectness of the encoded WF net. We essentially use programming language theory machinery, specifically that of type systems, to verify workflow nets.

3 A hypothetical language

3.1 Motivation

There is a need to justify the creation of an, albeit simple, programming language for our task. Our scheme requires a programming language with odd features:

1. Type declarations with no definitions: for use with associating with places.

2. Function declarations with no function body: for use with associating with transitions.
3. Bypassing projection of values of product types: for when there is an arrow from an AND-split transition to some place.
4. Bypassing pattern matching of values of sum types: for when there is an arrow from an OR-split transition to some place.

We need a programming language with these odd features, because to our knowledge, there is no language that has type declarations with no right-hand side definitions, functions with no definitions[1], and sum and product types but without pattern matching and projection functions, respectively.

3.2 Syntax

We call our hypothetical language H_{imp}. It has the following features:

1. is imperative and has only one (global) scope,
2. has product and sum types (with no mixing among them) but no projection functions and pattern matching,
3. has no `Unit` or `void` types, as transitions always have at least one input and at least one output so the functions that encode them always have non-zero arities,
4. has a first-order type system, meaning there is no type parametrization [3], e.g., there are no types like `List<T>`, and
5. has no higher-order functions because the WF nets it intends to encode don't have a notion of "transitions being inputs or results to and from other transitions".

The rest of the odd features mentioned in Section 3.1 is made apparent in the syntax of H_{imp} in Figure 5.

(Types)	α	$::=$	$T \mid \alpha_\times \mid \alpha_+$
	$\alpha_\times$	$::=$	$\alpha_{1\times} \times \alpha_{2\times}$
	α_+	$::=$	$\alpha_{1+} + \alpha_{2+}$
(Declarations)	D	$::=$	typedef T | fun $F\colon \alpha_1 \to \alpha_2$
(Assignments)	A	$::=$	var $I_s\colon T = val()$ | var $I\colon T = F(O)$
(Objects)	O	$::=$	$I' \mid O_1 \times O_2 \mid O_1 + O_2$
(Commands)	C	$::=$	$D; \mid A; \mid C_1\ C_2$

Fig. 5. Abstract syntax of H_{imp} in Backus-Naur Form.

Types are either non-composite types indicated by T (represented by upper-case letters in real code), product types $\alpha_\times$, or sum types α_+. The numbered

[1] The closest would be C forward declarations and Java interfaces, but those still require definitions written elsewhere in the code.

subscripts indicate that constituent types are distinct. The (Types) rules disallow types that mix product and sum types, for example, $G \times H + I$. This actually need not be explicitly stated in separate rules because encoded transitions don't mix AND- and OR-constructs on one side anyway, but constructs can still appear on both input and output sides, like in Figure 4, but this separation will be useful later in the type systems checking, found in Section 3.3.

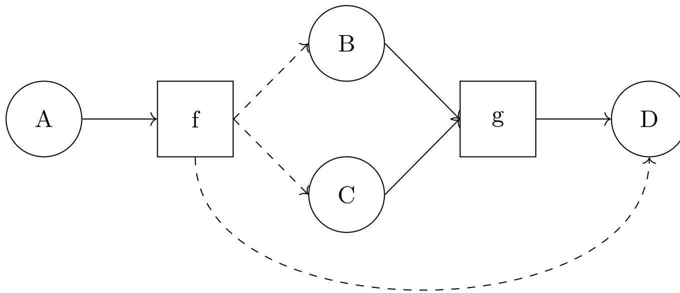
Declarations are either type definitions, or function declarations without a body but with explicit type annotations (F stands for function identifiers). The numbered subscripts indicate that output and input types must be different, as we limit ourselves to WF nets whose transitions have different input and output places.

Assignments to some variable with identifier I are either some special $val()$ assignment for giving value to the variable that encodes the start place, or some other assignment interpreted from arrows going into places. Functions (interpreted from arrows going into transitions) in this language are unary, but to accommodate multiple inputs encoded from AND-/OR-join constructs, the (Objects) rule allow for product and sum type value constructors in the forms $O_1 \times O_2$ and $O_1 + O_2$, respectively, to collect multiple variables into one argument. Notice that it's technically possible with the (Objects) rule to create objects with mixed constructors, for example, $g \times h + i$, but we don't need to add a rule separating these constructors because encoded transitions are only either AND-joins or OR-joins on the input side, never a mix of both.

Commands are either declarations, assignments, or their sequencing. Notice that this grammar technically admits programs that are only declarations, or only assignments. Further, it can also allow for multiple special assignments or multiple assignments to the same identifier. Programs that are "only declarations" would encode a net with places (type definitions) and transitions (function definitions) but *without arrows*; programs that are "only assignments" would encode a net with arrows but *without places and transitions*. Programs that allow for multiple special assignments mean that the WF net has more than one starting place, which makes it cease to be a workflow net by Definition 2. We won't be adding extra rules to our grammar to rule out these absurdities because we assume that our scheme only admits WF nets that are at least correctly constructed based on Definition 2, so the source code we get will have both declarations and assignments and only one special assignment. The feature that allows multiple assignments to the same identifier is unconventional in typical programming (that would be a redeclaration error in real programming languages), but for our purposes, this is expected as this piece of syntax encodes incoming arrows (possibly more than one) from transitions to places as discussed in Section 2.3.

We see this syntax in action in Figure 6.

For simplicity, variable identifiers are given the lowercase names of their types. For function calls that need to take in more than one argument, all its input are bundled into one object using either `x` or `+`; in Figure 6, we use `b x c`




```
typedef A;
typedef B;
typedef C;
typedef D;

fun f: A -> B + C + D;
fun g: B x C -> D;

var a: A = val();
var b: B = f(a);
var c: C = f(a);
var d: D = f(a);
var d: D = g(b x c);
```

Fig. 6. Example encoding of a WF net to H_{imp}.

as the input argument because function `g` encodes a transition with an AND-join on its input side.

3.3 Type checking

The WF net in Figure 6 actually runs into a deadlock. After transition f fires, it produces one token on either place B, C, or place D. If it ends up placing a token on D, the process is finished. But if it places a token on either B or C, transition g cannot fire because it only sees one token among its input places B and C, but it needs one per place so it can fire because it's an AND-join. This is a WF net that works only occasionally.

We need a formalism to rule out such WF nets by doing a static analysis on the H_{imp} code that they get encoded to. This section introduces that formalism in the form of a type system for H_{imp}.

Much of the discussion about the design of the type system will be that of handling product and sum types in the language. We deal with them first before writing out all the type rules.

Dealing with product types In real programming languages with product types, simply assigning a value of, say, type $B \times C$ to a variable of type B makes the type checker complain; there is a need to use the correct projection function on the value with type $B \times C$ to extract the value of the appropriate (atomic) type for successful assignment.

With H_{imp} having no projection functions, we can design its type system to consider the sample code fragment `var b: B = f1(a);` to be *valid*, so long as the type of function call `f1(a)` is a product type that contains a constituent type `B` somewhere in it. We do this to simplify H_{imp}'s syntax, and more importantly, so the encoding scheme doesn't need to correctly infer which projection function to use in variable assignments involving product types.

Dealing with sum types The problem with the WF net in Figure 6 is that a transition with an OR-split output connects to places that themselves connect to transitions with an AND-join input. The problem is not because an OR-split feeds into an AND-join, but that the AND-join has inputs that come out of the same OR-split. The deadlock results from the AND-join needing all input places to have a token but since these places are also outputs from an OR-split, not all of them gets a token. To resolve this, the type system can enforce a structural check on function declarations, such that for a function with a product type as its input type:

1. all of its constituent input types must not show up in output sum types of other function declarations, or
2. if some of its constituent types show up in output sum types of other function declarations, for all pairings, the two types must not belong to the same output sum type of another function declaration.

In Figure 6, function g has constituent input types B and C, all of which belong to the output sum type $B+C+D$ of f. Looking at this unordered pair $\{B, C\}$, we see that $B \in \{B, C, D\} = \{B, C, D\} \ni C$; this violates our structural rule, which we want, so WF nets like in Figure 6 are ruled out.

The essence of this rule makes sure that sum types don't pose a problem for AND-joins; an AND-join requires all of its input places to have a token, so if any pair of these places cannot guarantee a token each, i.e., they are part of the output places of the same OR-split somewhere, then the AND-join cannot fire.

This rule still allows for AND-joins having some input places that are also outputs from OR-splits, so long as those places don't belong to the same OR-split output. We see this in Figure 7.

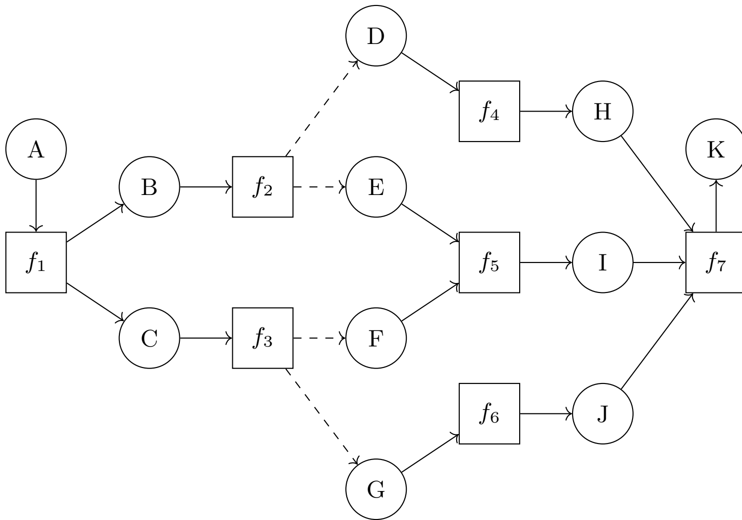


Fig. 7. A WF net with OR-splits followed by an AND-join.

The OR-splits f_2 and f_3 feed into an AND-join f_5. The constituent input types of function f_5, type E and type F, all belong to output sum types of f_2 and f_3, respectively. Looking at all the unordered pairs of the constituent input types of f_5, we get the singleton set $\{\{E, F\}\}$, and examining it, we find that $E \in \{D, E\} \neq \{F, G\} \ni F$, so the rule is not violated.

For f_7, none of its constituent input types H, I, and J show up in other functions' output sum types respectively, so the rule is automatically not violated, no further pairwise checking required. We say that this WF net passes our check.

This, however, is not enough. Simulating the execution of the WF net in Figure 7, we find that it still runs into a deadlock. f_1 is an AND-split so OR-splits f_2 and f_3 will always fire. If there are tokens in E and F, only place I gets the token, thus f_7 won't fire. If there are tokens in D and F, only place H gets

a token, f_7 won't fire. Symmetrically, if there are tokens in E and G, f_7 also won't fire. At best, if there are tokens in D, and G, places H and J get a token, but it isn't enough to trigger f_7.

3.4 Another static check

We need another check that deals with the effects of an OR-split somewhere in a WF net. For this, we take another inspiration from programming languages, in the form of *nullable types.*

Definition 6. *A nullable type, denoted by $T?$, refers to a type whose values are either some special null value, or the usual values of the type T.*

We adopt the idea of nullable types into our work with WF nets. In this context, where places get encoded as types, we say that a nullable type encodes a place where a token may or may not appear, due to it being an output place for some OR-split transition. We say that a place's *nullability*, its uncertainty of having a token, is *infectious*, because if a place is nullable, i.e., it may not have a token, then any transition it feeds into, inherits the possibility of not firing, thus the output places of that transition end up nullable, too, and so on.

Definition 7. *Infectious nullability refers to the property of a place in which it may or may not have a token. It is infectious, as this uncertainty propagates forward into transitions that the place points to (uncertainty of firing), and to the places those transitions point to (uncertainty of having a token), and so on. A place acquires this property if it is an output place of an OR-split transition.*

Given a WF net with at least one OR-split transition, and using Definition 7, we get a portion of the places of this WF net that are nullable. We now setup another encoding:

1. Nullability is the uncertainty of having a token or not; we encode this as Boolean values true or false, respectively, i.e., true means "has token".
2. Nullable places can have a token or not; we encode them as atomic Boolean variables.
3. Transitions are encoded as equal signs (like in an equation).
4. AND-joins fire when all its input have tokens; we encode them as conjunction to the left of equal signs.
5. OR-joins fire when at least one of its inputs has a token; we encode them as disjunction to the left of equal signs.
6. AND- and OR-splits get encoded as conjunction and disjunction, respectively, to the right of equal signs.

Using this encoding, and a WF net with at least one OR-split transition, we can get a set of Boolean equations. We now check if all valid Boolean assignments to the initial nullable places, i.e., those that started the propagation of infectious nullability, result to the Boolean equations giving the atomic proposition that

encodes the end place, a value of true; that is, we check using the Boolean equations if the end place ends up having a token, regardless of the token conditions that the OR-splits produce.

We use this "Boolean equation check" for Figure 7. OR-splits f_2 and f_3 produce the initial nullable places D, E, F, and G. From these, the nullability propagates forward, giving us the following Boolean equations:

1. $D = H$
2. $E \wedge F = I$
3. $G = J$
4. $H \wedge I \wedge J = K$

We setup a truth table for the initial nullable places, taking into account their relationships, e.g., due to the OR-split in f_2, which we interpret to be XOR in Section 2, D and E cannot be both true, and cannot be both false at the same time, and the same goes for F and G due to f_3. This truth table is completed according to the Boolean equations above, as seen in Table 1.

D	**E**	**F**	**G**	**H**	**I**	**J**	**K**
T	F	T	F	T	F	F	F
F	T	T	F	F	T	F	F
F	T	F	T	F	F	T	F
T	F	F	T	T	F	T	F

Table 1. Completed truth table for the nullable places of the Boolean equation check for Figure 7.

See that K gets a false on all valid Boolean assignments on D, E, F, and G; place K will never get a token, and we say that this WF net will always deadlock.

Infectious nullability always propagates up to the end place (implied by strong connectedness property in Definition 2), so we can use the Boolean equation check to conclude something about the possibility of the end place getting a token whenever there is at least one OR-split in a WF net.

3.5 Full type system

Type systems of typical programming languages often feature *environment extensions* of the form $\Gamma, x : \tau \vdash \xi$ found in the premises of some type rules [3], where x is some variable of type τ that gets added to the environment Γ to handle type checking when entering some local scope, e.g., inside a function. H_{imp} doesn't have local scopes because its functions have no bodies, thus, all programs only have one global scope, making the context Γ unchanged all throughout derivations when type checking.

When starting a derivation, the context Γ is already filled up with types, function identifiers and their types, and variables and their types. In the initial processing of the source code, we assume that the types in *typedef* declaration commands are distinct and thus they are not subject to type checking (but that they've already contributed to Γ).

Since the language is command-based, commands are not given types[2], so one judgment in the type system is simply of the form $\Gamma \vdash C$, which means that the command C is a well-formed command in Γ. Another judgment is of the conventional $\Gamma \vdash M : A$ form, where M could be a variable or a function identifier, and A could be an atomic type or a composite type, and overall it means that M is well-typed with type A in Γ. Another judgment is of the form $\Gamma \vdash T$, which says that type T is in Γ.

The central concern in the type checking of H_{imp} programs is that of function declarations with a product type input possibly being affected by other function declarations with a sum type output as discussed in Section 3.3. It is therefore helpful to introduce some notation for searching the context for the set of all sets of constituent types of functions with a sum type on the output side. Let's call it $sumout(\Gamma)$. As an example, using this on the WF net in Figure 7 yields $\{\{D, E\}, \{F, G\}\}$.

We need a shorthand for extracting the constituent types of composite data types into sets, and for this, we use $\{\alpha\}$. For example, $\{A \times B \times C\} = \{A, B, C\}$. Following this, we also need a shorthand for asserting that the constituent types of a composite data type are valid in the context, i.e., each exists in the context, and for this we use $\Gamma \ \vdash \{\alpha\}$ in the type rules, but when doing the actual derivation for checking the existence of types of a function with type $A \to B \times C$, for example, we'll have the following judgments: $\Gamma \vdash A$, $\Gamma \vdash B$, and $\Gamma \vdash C$. If this notation is applied to an atomic type, it just produces a singleton set containing that one atomic type.

The full type system is shown in Figure 8.

In the rule (Decl Func Prod Input), the long side condition encodes the structural check on function declarations with product type inputs as discussed in Section 3.3. The first clause flattens the set of sets that $sumout(\Gamma)$ produces with big union ($\bigcup$) and makes sure that no constituent type of the input product type $\alpha_\times$ is found in sum type outputs of other functions. Failing that, the pairwise check condition in the second clause is tried to see if no pairing of constituent types belong to the same sum type outputs of other functions. If it happens that an input constituent type is *not* a member of another sum type output while another input constituent type is a member of another sum type output of other functions, then one of the temporary sets in the second clause will end up being $\varnothing$ (because it can't find a set for which the constituent type belongs to since it doesn't appear in other sum type outputs), and then for any non-empty set A, $A \neq \varnothing$ is always true.

[2] In some real programming languages where commands and expressions are present, commands may be given the `Unit` type, e.g., F#, OCaml.

$$\frac{T \in \Gamma}{\Gamma \vdash T} \qquad \text{(T-type)}$$

$$\frac{I : T \in \Gamma}{\Gamma \vdash I : T} \qquad \text{(T-var)}$$

$$\frac{F : \alpha \to \alpha' \in \Gamma}{\Gamma \vdash F : \alpha \to \alpha'} \qquad \text{(T-func)}$$

$$\frac{\Gamma \vdash O_1 : T_1 \qquad \Gamma \vdash O_2 : T_2}{\Gamma \vdash O_1 + O_2 : T_1 + T_2} \qquad \text{(Sum Val Constr)}$$

$$\frac{\Gamma \vdash O_1 : T_1 \qquad \Gamma \vdash O_2 : T_2}{\Gamma \vdash O_1 \times O_2 : T_1 \times T_2} \qquad \text{(Prod Val Constr)}$$

$$\frac{\Gamma \vdash I : T \qquad \Gamma \vdash F : \alpha \to \alpha' \qquad \Gamma \vdash O : \alpha \qquad T \in \{\alpha'\}}{\Gamma \vdash \texttt{var}\ I : T = F(O)} \qquad \text{(Assign Var)}$$

$$\frac{\Gamma \vdash I_s : T}{\Gamma \vdash \texttt{var}\ I_s : T = \texttt{val()}} \qquad \text{(Assign Var Spec)}$$

$$\frac{\Gamma \vdash \{\alpha\} \qquad \Gamma \vdash \{\alpha'\} \qquad (\alpha \text{ is atomic or sum type})}{\Gamma \vdash \texttt{fun}\ F : \alpha \to \alpha'} \qquad \text{(Decl Func Non-Prod Input)}$$

$$\frac{\begin{array}{c} \Gamma \vdash \{\alpha_\times\} \\ \Gamma \vdash \{\alpha'\} \\ \forall\, T \in \{\alpha_\times\}. T \notin \bigcup sumout(\Gamma) \vee \\ \forall\, T_1, T_2 \in \{\alpha_\times\}. T_1 \neq T_2 \implies \\ \text{let } A := \{S \in sumout(\Gamma) \mid T_1 \in S\}, \\ B := \{S \in sumout(\Gamma) \mid T_2 \in S\},\ A \neq B \end{array}}{\Gamma \vdash \texttt{fun}\ F : \alpha_\times \to \alpha'} \qquad \text{(Decl Func Prod Input)}$$

$$\frac{\Gamma \vdash C_1 \qquad \Gamma \vdash C_2}{\Gamma \vdash C_1\ C_2} \qquad \text{(Comm Sequence)}$$

Fig. 8. Full type system of H_{imp}.

In the rule (Assign Var), assigning a value of composite type α' to a variable of type T is allowed if T is a constituent type of α'. This allows for statements like `var b: B = f(A)` where `f(A)` has type $B \times C$ as discussed in Section 3.3. Note that this also allows for statements like `var b: B = f(A)` where `f(A)` has sum type $B + C$ instead; this is all right because when sum types get involved, the "Boolean equation check" formalized in Section 3.4 can be used as the next step in static checking aside from type checking.

Figure 9, Table 2, and Figure 10 show a fully worked out example of the usage of the scheme, from a WF net representation and its H_{imp} source code, to type checking derivation, and Boolean checking (since an OR-split exists in the WF net), respectively. To save space, the derivation tree for the type checking are be split by command, i.e., only subtrees are shown per function declaration or per assignment, and applied (non-axiom) type rule names are shown in acronyms due to horizontal space constraints (if space is not a problem, the rule (Comm Sequence) would yield a tree that grows bigger towards the upper right as the metavariable C_2 gets expanded recursively with more and more commands). Remember that the context Γ is unchanging and is always of value

$$\{A, B, C, D, E, F, G, H, I,$$
$$f_1 : A \to B \times C, f_2 : B \to D, f_3 : C \to E + F,$$
$$f_4 : D \times E \to G, f_5 : F \to H, f_6 : G \times H \to I,$$
$$a : A, b : B, c : C, d : D, e : E, f : F, g : G, h : H, i : I\}$$

in this example.

Table 2: Derivation for the H_{imp} program of the net in Figure 9.

$$\frac{\overline{\Gamma \vdash A} \quad \overline{\Gamma \vdash B} \quad \overline{\Gamma \vdash C}}{\Gamma \vdash \texttt{fun}\ f_1 : A \to B \times C} \ (\mathsf{DFNI})$$

$$\frac{\overline{\Gamma \vdash B} \quad \overline{\Gamma \vdash D}}{\Gamma \vdash \texttt{fun}\ f_2 : B \to D} \ (\mathsf{DFNI})$$

$$\frac{\overline{\Gamma \vdash C} \quad \overline{\Gamma \vdash E} \quad \overline{\Gamma \vdash F}}{\Gamma \vdash \texttt{fun}\ f_3 : C \to E + F} \ (\mathsf{DFNI})$$

$$\dfrac{\overline{\Gamma \vdash D} \quad \overline{\Gamma \vdash E} \quad \overline{\Gamma \vdash G} \quad \varnothing \neq \{E, F\} \ni E}{\Gamma \vdash \texttt{fun}\ \ f_4 : D \times E \to G} \text{ (DFPI)}$$

$$\dfrac{\overline{\Gamma \vdash F} \quad \overline{\Gamma \vdash H}}{\Gamma \vdash \texttt{fun}\ \ f_5 : F \to H} \text{ (DFNI)}$$

$$\dfrac{\overline{\Gamma \vdash G} \quad \overline{\Gamma \vdash H} \quad \overline{\Gamma \vdash I} \quad G, H \notin \{E, F\}}{\Gamma \vdash \texttt{fun}\ \ f_6 : G \times H \to I} \text{ (DFPI)}$$

$$\dfrac{\overline{\Gamma \vdash a : A}}{\Gamma \vdash \texttt{var}\ \ a : A = \texttt{val()}} \text{ (AVS)}$$

$$\dfrac{\overline{\Gamma \vdash b : B} \quad \overline{\Gamma \vdash f_1 : A \to B \times C} \quad \overline{\Gamma \vdash a : A} \quad B \in \{B, C\}}{\Gamma \vdash \texttt{var}\ \ b : B = f_1(a)} \text{ (AV)}$$

$$\dfrac{\overline{\Gamma \vdash c : C} \quad \overline{\Gamma \vdash f_1 : A \to B \times C} \quad \overline{\Gamma \vdash a : A} \quad C \in \{B, C\}}{\Gamma \vdash \texttt{var}\ \ c : C = f_1(a)} \text{ (AV)}$$

$$\dfrac{\overline{\Gamma \vdash d : D} \quad \overline{\Gamma \vdash f_2 : B \to D} \quad \overline{\Gamma \vdash b : B} \quad D \in \{D\}}{\Gamma \vdash \texttt{var}\ \ d : D = f_2(b)} \text{ (AV)}$$

$$\dfrac{\overline{\Gamma \vdash e : E} \quad \overline{\Gamma \vdash f_3 : C \to E + F} \quad \overline{\Gamma \vdash c : C} \quad E \in \{E, F\}}{\Gamma \vdash \texttt{var}\ \ e : E = f_3(c)} \text{ (AV)}$$

$$\dfrac{\overline{\Gamma \vdash f : F} \quad \overline{\Gamma \vdash f_3 : C \to E + F} \quad \overline{\Gamma \vdash c : C} \quad F \in \{E, F\}}{\Gamma \vdash \texttt{var}\ \ f : F = f_3(c)} \text{ (AV)}$$

$$\dfrac{\overline{\Gamma \vdash g : G} \quad \overline{\Gamma \vdash f_4 : D \times E \to G} \quad \dfrac{\overline{\Gamma \vdash d : D} \quad \overline{\Gamma \vdash e : E}}{\Gamma \vdash d \times e : D \times E}\text{(PVC)} \quad G \in \{G\}}{\Gamma \vdash \texttt{var}\ \ g : G = f_4(d \times e)}\text{(AV)}$$

$$\dfrac{\overline{\Gamma \vdash h : H} \quad \overline{\Gamma \vdash f_5 : F \to H} \quad \overline{\Gamma \vdash f : F} \quad H \in \{H\}}{\Gamma \vdash \texttt{var}\ \ h : H = f_5(f)}\text{(AV)}$$

$$\dfrac{\overline{\Gamma \vdash i : I} \quad \overline{\Gamma \vdash f_6 : G \times H \to I} \quad \dfrac{\overline{\Gamma \vdash g : G} \quad \overline{\Gamma \vdash h : H}}{\Gamma \vdash g \times h : G \times H}\text{(PVC)} \quad I \in \{I\}}{\Gamma \vdash \texttt{var}\ \ i : I = f_6(g \times h)}\text{(AV)}$$

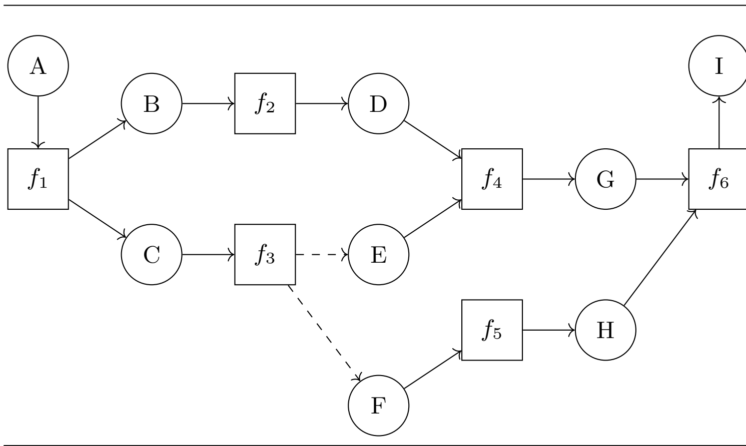
4 Correctness criteria

The standard correctness requirements for WF nets, called *classical soundness*, are the following [1]:

1. option to complete,
2. proper completion, and
3. no dead transitions

The first requirement says that there exists a sequence of firings of transitions such that one token in the start place eventually places a token in the end place. Our scheme can check for this, specifically the Boolean check. In Figure 6 for example, the resulting truth table would be a mix of true and false for place D: true if f places a token on D immediately, and false if f places a token on either B or C, which leads to a deadlock. Additionally, the Boolean check can determine if deadlocks completely prevent completion for all possible token configurations of places affected by OR-splits.

The second requirement says that when the end place gets a token, all other places must be empty. This is not guaranteed by the scheme. One way to work towards this is by adding substructural features to the type system, specifically, making a *linear* type system. A linear type system ensures that every variable is used exactly once [5]. In the current scheme, this isn't immediately possible because variables can be used more than once, for example, in Figure 6, variable a is used twice to produce values for variables b and c. Still, the WF net action of consuming a token from a place can be interpreted as the usage of a variable and if this usage is restricted to happen exactly once using a linear type system,




```
typedef A;
typedef B;
typedef C;
typedef D;
typedef E;
typedef F;
typedef G;
typedef H;
typedef I;

fun f1: A -> B x C;
fun f2: B -> D;
fun f3: C -> E + F;
fun f4: D x E -> G;
fun f5: F -> H;
fun f6: G x H -> I;

var a: A = val();
var b: B = f1(a);
var c: C = f1(a);
var d: D = f2(b);
var e: E = f3(c);
var f: F = f3(c);
var g: G = f4(d x e);
var h: H = f5(f);
var i: I = f6(g x h);
```

Fig. 9. Another WF net encoded to H_{imp}, with derivation found in Figure 2.

Boolean equations:

$$D \wedge E = G$$
$$F = H$$
$$G \wedge H = I$$

Truth table if E is true and F is false, i.e.,
if transition f_3 happens to produce a token at E:

D	**E**	**F**	**G**	**H**	**I**
T	T	F	T	F	F

Truth table if E is false and F is true, i.e.,
if transition f_3 happens to produce a token at F:

D	**E**	**F**	**G**	**H**	**I**
T	F	T	F	T	F

Fig. 10. Boolean checking for the H_{imp} program of the WF net in Figure 9. Note that D is always true because its 'predecessor', B, comes from an AND-split, which always produces tokens on all output places, which we interpret as Boolean true as discussed in Section 3.4. In any case, looking at the truth value at I, which is false in all scenarios, we conclude that this WF net deadlocks and won't work even occasionally.

the second requirement can be realized. As for assignments coming from function calls, destructuring might have to be introduced to the language to keep variable usages in function calls strictly one-and-unique.

The third requirement says that any arbitrarily chosen transition can be fired following an appropriate route through the WF net. This is different from the strong connectedness property of the WF net itself when a transition is connected from the end place to the start place according to Definition 2, as this property is structural, while the third requirement is a behavioral property. An analog in terms of programming languages is that, this scheme is a check that happens at compile time, while checking for the third requirement is some runtime effort. The scheme doesn't cover this runtime/behavioral check.

With this, we say that the scheme verifies WF nets up to *lazy soundness*, which is defined as a soundness that focuses only on the end place and allows excess tokens in other places [1].

5 Conclusion and future work

In this paper, we observe that a WF net can be encoded as source code of a statically typed programming language. For this purpose, we create a hypothetical programming language H_{imp}, and we conjecture that its successful static checking entails a certain correctness of the encoded WF net. The static check is done in two stages at most: (1) a typical type checking phase, and (2) a Boolean

equations check that is derived from the places in the WF net that follows from an OR-split transition, if it exists. The scheme can check up to lazy soundness.

The scheme has the following limitations. It doesn't admit WF nets with iteration in them, as otherwise, following a naive encoding would lead to non-termination, because variable assignment would happen indefinitely, manifested as variable redeclarations that never end. The simplicity of the type system also can't enforce the 'proper completion' requirement of classical soundness; the modification of the type system into a linear type system can help strengthen the soundness the scheme can check.

Proving the consistency of the proposed type system is beyond the scope of this paper. For this, a *type soundness theorem* have to be proven [3]. The common approach is the proving of *progress* and *preservation* theorems [6], but this requires a notion of evaluation in the language, typically via small-step operational semantics. The crafting of operational semantics for H_{imp} that corresponds to WF net semantics is also beyond the scope of this paper.

Future work include (1) formalizing the encoding scheme to a proper algorithm with some representation of the WF net as input and H_{imp} code as output, (2) modifying the type system to be a linear type system so it becomes capable of checking for leftover tokens, (3) proving the internal consistency of the type system to avoid absurd behavior for well-typed programs, and (4) tying a type checking algorithm to the type system, so the complexity of the entire scheme can be compared with existing methods for checking WF net soundness.

Acknowledgments

I would like to thank Dr. Richelle Ann Juayong and Dr. Francis George Cabarle, my professors in a course on workflow models, and Dr. Jasmine Malinao for their assistance.

References

1. Aalst, W., Hee, K., Hofstede, A., Sidorova, N., Verbeek, H., Voorhoeve, M. & Wynn, M.: Soundness of workflow nets: Classification, decidability, and analysis. *Formal Aspects Of Computing.* **23**, 333-363 (2011,5)
2. Aalst, W. & Hee, K.: Workflow management: Models, methods, and systems. (MIT Press, 2004)
3. Cardelli, L.: Type systems. *ACM Computing Surveys.* **28**, 263-264 (1996,3)
4. Aalst, W.: Verification of workflow nets. *Lecture Notes In Computer Science.* pp. 407-426 (1997)
5. Pierce, B.: Advanced topics in types and programming languages. (MIT Press,2010)
6. Pierce, B.: Types and Programming Languages. (MIT Press,2002)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial 4.0 International License (http://creativecommons.org/licenses/by-nc/4.0/), which permits any noncommercial use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.