Mauro Gambini

The Design of Graphical Process Modeling Languages: from Free Composition to Modular Construction

Ph.D. Thesis

September 18, 2012

Università degli Studi di Verona
Dipartimento di Informatica

Advisor:
prof. Carlo Combi

Series N°: **TD-05-12**

Università di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona
Italy

to my wife Sara

Abstract. A graphical Process Modeling Language (PML) is a language tailored for modeling software systems by means of process models. It is said to be graphical because the primary representation of models are diagrams obtained combining visual constructs and previously defined components. Graphical PMLs are interesting as they open the design space to new geometric representations of complex interrelated aspects like concurrency and interaction. A Process-Aware Information System (PAIS) is a software system driven by explicit process models with the aim to coordinate and support agents in performing their activities. It is responsible for managing several process model instances at the same time balancing the available resources. A PML is the primary interface of a PAIS and a main concern in its design, because it is used by end-users, consultants, and developers for understanding, implementing and enacting complex processes. The adoption of PAIS technology may be severely limited by the weakness of PMLs in describing complex use cases.

The overall aim of this thesis is to improve the design of graphical PMLs in order to engineer more effective PAISs. This goal is pursued following three intertwined paths: firstly, mainstream PMLs and their theoretical foundations are analyzed for exposing their features and limits; secondly, a widespread PML verification method is consolidated and then extended with a novel technique for automating process correction; finally, an alternative PML design solution is explored through a proof-of-concept language, called NestFlow, that improves both modularity and comprehensibility by providing a more structured modeling approach. A modular approach is only possible if data-flow dependencies are accepted as a main concern in PML design. NestFlow tries to ease the modeling activity by providing a comprehensive set of tightly integrated control-flow and data-flow constructs, promoting the latter as first-class citizens in process modeling.

Acknowledgement

I would like to thank my advisor, prof. Carlo Combi, for his help and involvement in my PhD course.

I am also grateful to dr Marcello La Rosa and prof. Arthur ter Hofstede who hosted me and collaborated with me during and after my visit at Queensland University of Technology, Brisbane, Australia.

I would also like to thank my PhD thesis referees prof. Manfred Reichert, prof. Marlon Dumas, and prof. Giuseppe Pozzi for their precious comments on my work.

The greatest thank goes to Sara, who collaborated with me and encouraged me during all these years.

Contents

1

Introduction

The notion of process is fundamental in science and engineering: the primary activity of all sciences is extracting new knowledge about a certain process by proposing a model of it and conceiving experiments to confirm or disprove its validity. Engineering may be seen as a problem-solving activity focused on the control, support, and automation of processes. Broadly speaking, a *process* can be defined as a series of occurring changes in a system that span over time producing some measurable effects. Any non-trivial process involves several entities that simultaneously change their configuration and interact together following certain patterns. An explosion, for example, is an highly concurrent process with a massive number of atomic interactions that occur in a fraction of time. Undoubtedly, any specific domain has its own notion of process, but it is hard to imagine a definition of process decoupled from the concepts of *time* and *state*. It is also hard to explain the emerging *behavior* of a process without considering how the involved entities interact and when the relevant events occur.

From a software engineering perspective, computer programs are crafted to shape the behavior of machines that will be part of more or less complex processes in the physical world [1]. These machines have to deal with the inherent concurrency of such world and they have to interact with it to be of some help. Not surprisingly, programming languages reflect this fact by offering specific language constructs and special libraries of components that allow one to capture the aforementioned concepts. Processes may also be adopted as basic building components for designing concurrent interactive software systems. In such case the role of processes is similar to the role of objects in the Object-Oriented Programming (OOP) paradigm: OOP is based on the analogy between building a mechanical model of a physical system from concrete objects and building a software model of a physical system from software components [2]. Let us stress that there is no a one-to-one correspondence between real entities and software components: recalling the example of Abadi and Cardelli [2], a mechanical model of the solar system may contain objects like springs and gears that are not part of the modeled reality.

A *Process Modeling Language (PML)* is a specialized language tailored for modeling software systems in terms of processes; therefore, it has to offer a native support for specifying the logic of components that can run in parallel and interact among them and with the environment. The language is said to be *specialized* to

emphasize that it is designed around few basic and precisely stated abstractions in contrast to a more rich and ambiguous natural language. The term *formal* is intentionally avoided because it is more suitable for qualifying mathematical languages with a clearly stated formal semantics, and not all PMLs are formal.

A specialized language becomes necessary in unraveling complex systems where several people have to agree upon a common design. Furthermore, if the produced models are sufficiently formal, they can be simulated, verified, translated, and also directly interpreted by machines. These considerations explain why *modularity* cannot be neglected in the analysis of modeling languages that are, in their essence, tools for tackling the inherent complexity of the design activity. Modularity is the property of a system to be decomposed into smaller interrelated parts which can be recombined in different configurations [3]. This notion can be applied to a PML that is said to be modular if its models can be decomposed into smaller reusable components which in turn can be recombined for exploring different design alternatives. Modularity reduces the efforts needed to change a system; hence, it improves its flexibility, i.e. the ability of adapting to new contingent needs [4].

This thesis focuses on *graphical* PMLs. A modeling language is said to be *graphical* if the primary representation of a model is a diagram obtained combining visual symbolic constructs and previously defined components. This is in contrast with purely textual languages that allow one to express models only as a sequence of symbols. Graphical PMLs are interesting because they open the design space to new geometric representations of complex interrelated aspects like concurrency and interaction. This feature should not be underestimated because reasoning about concurrent entities is notoriously a very difficult cognitive activity: it is well-known, for example, that multi-threaded programs with shared state become soon incomprehensible to humans [5].

Context

From one hand, the thesis attempts to be neutral about PMLs, in the sense that it focuses on the abstractions offered by a language and how they are implemented, without any particular emphasis on their domain-specific meaning. At first glance, for example, every language has its own notion of component that may be called task, activity or workflow, to mention few names, but behind some implementation details they are often realized in the same way.

On the other hand, the thesis is mostly centered on PMLs used to design and implement information systems, more precisely on the theoretical and practical languages adopted in Process-Aware Information Systems (PAISs) [6] that broadly speaking include Business Process Management Systems (BPMSs) [7] and Workflow Management Systems (WfMSs) [8].

A *Process-Aware Information System (PAIS)* [6] is a software system driven by explicit process models with the aim to coordinate and support agents in performing their activities. In this context a process is a collection of interrelated activities that are performed in an organizational and technical environment for achieving a predefined goal [7]. Accordingly, a PML can be classified as a coordination language [9] for human and software agents, where classical computations have a secondary role with respect to concurrency, interaction, and integration.

Motivation

A PML is the primary interface of a PAIS as it is used by consultants, developers and end-users: consultants can model existing or desired processes with the end-users for understanding how the work is carried out. Developers can enrich such models with further details and missing components for implementing the actual system. End-users will run several instances of different process models to streamline their activities. A PML is also a primary concern in the design of a PAIS, because most of its behavior is determined by the interpreted models than have to specify in some way how process instances, data and resources are managed.

The overall aim of this thesis is to improve the design of graphical executable PMLs in order to engineer more effective PAISs that better match the aforementioned vision. This goal is pursued following three intertwined paths that mostly correspond to the three central chapters of this thesis: Firstly, mainstream PMLs and their theoretical foundations are analyzed in order to expose their features and limits. Secondly, a widespread PML verification method is consolidated and then extended with a novel technique for automating process correction, called Petri Nets Simulated Annealing (PNSA). Finally, an alternative PML design solution is explored through a prototypical modeling language, called NestFlow, that leaves the widely accepted free-composition paradigm in favor of a more structured approach that in turn enhances modularity and comprehensibility.

A modular approach is only possible if data-flow dependencies are accepted as a main concern in PML design. Therefore, NestFlow promotes data-flow constructs as first-class citizens in process modeling. Conversely, despite some notable exceptions like artifact-centric approaches [10] and case-handling [11], PMLs usually focus on control-flow modeling: they provide a wide range of graphical constructs for specifying the ordering relations among components, but very few means for representing data dependencies. Data are considered implementation details that have to be finally added to the specified control-flow structures in order to obtain an executable model. This practice helps in obtaining simpler PMLs, but actually there is no evidence that simple languages ease the modeling activity. For example, Workflow Control-Flow Patterns (WCPs) [12], that capture recurring process behaviors, are specified in high-level Coloured Petri Nets (CPNs) [13], not in the simpler ordinary Petri Nets [14]. After the introduction of WCPs, some domain-specific PMLs, like YAWL [15], have been designed to express some patterns hard to obtain in CPNs. NestFlow tries to overcome these difficulties and to ease the modeling activity by providing a comprehensive set of tightly integrated control-flow and data-flow constructs.

Contribution

The first contribution of this thesis is a comprehensive and uniform analysis of the state of the art about graphical PMLs and their theoretical foundation. This analysis includes representative languages chosen from formal modeling languages, academic research projects and industrial standards. In particular, it focuses on Place Transition Nets (PTNs) [14], high-level Coloured Petri Nets (CPNs) [13], Yet Another Workflow Language (YAWL) [15], and Business Process Model and

Notation (BPMN) [16]. This chapter can be considered a detailed survey about the essential design principles and features of these languages and the existing relations among them.

Available graphical PMLs mostly adopt a free-composition paradigm accordingly to which components and constructs can be put together by connecting them with arrows of various forms, subject to very few syntactical constraints. Structure cannot be enforced in these languages without reducing their expressiveness [17]. As a consequence, the resulting models are usually unstructured and are likely to contain subtle errors that need to be detected and corrected. The first contribution of Chap. 4 is a revisited notion of soundness that better classifies errors found in a model. Based on this new definition, a refined version of the soundness check procedure is introduced and explained. This procedure is an important component of the novel technique called Petri Nets Simulated Annealing (PNSA) that has been published in [18], and is discussed in the remaining part of the chapter. PNSA is a genetic programming technique inspired by Multi-Objective Simulated Annealing methods. Given an unsound model and the results of the soundness check, PNSA searches for a set of solutions that are structurally and semantically similar to the original model but contain few errors.

Another contribution of the thesis is a detailed analysis of the limits underlying unstructured PMLs and their adopted free-composition paradigm. This analysis reveals that many arguments supporting such paradigm are not well founded, and they are so widespread to prevent any further investigation about alternative approaches. This analysis has been published in [19,20] and deeply examined in Chap. 5. This discussion leads to the development of NestFlow: an innovative prototypical PML able to support a block-structured control-flow design with positive effects for modularity and comprehensibility of models.

The NestFlow expressiveness is evaluated against the well-known workflow control-flow patterns framework [12], but using a more objective evaluation method which considers the effort needed to replicate the behaviour prescribed by a pattern, rather than the availability of a particular language construct. The evaluation method and the obtained results are published in [21] and further discussed in Chap. 6. This chapter also presents two additional contributions published in [22] and [23], which discuss the NestFlow applicability and extensions in the geographical and health-care domains, respectively.

Organization

Chapter 2 – Background. This chapter introduces the mathematical notation adopted through the entire thesis to express concepts in a formal way. The chapter also introduces basic mathematical structures that are used in the following chapters and show how concepts can be encoded in the given functional-like notation.

Chapter 3 – Graphical Process Modeling Languages. This chapter is a comprehensive introduction to PMLs and their related theoretical foundation. It focuses on four representative languages, namely Place Transition Nets (PTNs) [14], Coloured Petri Nets (CPNs) [13], Yet Another Workflow Language (YAWL) [15], and Business Process Model and Notation (BPMN) [16]. Each language is introduced in

an informal way, and then its essential features are analyzed in more formal terms.

Chapter 4 – Free Composition, Verification and Correction. Initially, the classical soundness definition is discussed and some issues about it are exposed, then a revisited formalization is provided which ensures orthogonality of its characterizing properties. This gives the opportunity to refine the existing soundness techniques to produce more useful information about the detected errors. Subsequently, the problem of automating the correction of the found errors is considered. The remaining part of the chapter discusses the PNSA solution together with some preliminary experimental results.

Chapter 5 – Towards Structured Process Modeling Languages. The chapter starts by showing what goes wrong with existing PMLs: pitfalls of unstructured process modeling and myths surrounding unstructured modeling languages are deeply discussed with some examples. Then the chapter argues why structure is fundamental for model comprehensibility, and it exposes the lack of modularity of certain PMLs. These problems justify the search for alternative approaches in PML design. Accordingly, an innovative prototypical PML, called NestFlow, is proposed: this language provides block-structure control-flow abstractions with message passing, boosting up both modularity and comprehensibility.

Chapter 6 – NestFlow Expressiveness and Applications. The chapter discusses the expressiveness of NestFlow in terms of workflow control-flow patterns and its potential application in different domains, such as business process automation, geo-processing, and health-care information systems. This analysis suggests that a structured modeling language can be effectively built and applied for the modeling of real processes.

Chapter 7 – Conclusion. The chapter summarizes the results presented in the thesis and proposes future work.

2

Background

This chapter introduces the mathematical notation adopted in the following to express concepts in a formal way. The notation is introduced here by discussing basic mathematical notions and because it is fairly intuitive: the reader should feel free to skip this part and come back later if some formal statement is not clear.

An *object* is any entity recognized as a single unit. A *set* is a group of objects called *elements* that can be identified as a single unit, hence a set can be an element of another set. It is common to represent objects with unique *names* or *identifiers* and use them as elements to build abstract sets. An identifier that represents a specific object is said to be a *constant* or *value*, while an identifier that stands for an unknown object is called *variable*.

A different typeface is adopted to distinguish constants from variables whenever necessary. In particular, the *italic* typeface is reserved for variables, while constants and values are usually expressed in sans serif typeface or in its SMALL CAPS variant depending on the context. Furthermore, as a general convention, lowercase letters like a, x, σ are used to identify elements, while uppercase letters like A, X, Σ are reserved for sets. This distinction should not be intended as a strict rule because sets are also elements. When necessary, both kinds of identifiers are extended with subscripts and superscripts, e.g. x_i and A_k are also valid identifiers.

A finite set can be explicitly built listing its elements between curly braces { and }, where both the presence of duplicates and the order in which elements appear is not relevant, e.g. $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$ and $\{\mathsf{B}, \mathsf{C}, \mathsf{B}, \mathsf{A}\}$ are the same set made of three symbols representing the first letters of the alphabet. From the possible set structures, one can distinguish the *empty set* $\{\}$, denoted by the special symbol $\varnothing$, and the set $\{x\}$ made of a single element x known as *unit set* or *singleton*. An *ordered pair* or simply a *pair* is a set of two elements constructed in such a way that it is always clear which are the former and the latter. For every two elements x and y the set $\{x, \{x, y\}\}$ has this property and will be denoted as $\langle x, y \rangle$.

The usual *membership* operator $\cdot \in \cdot$ is adopted to state that an element is a *member of* or *belongs to* a set. In particular, for any variable x and any set S, the term $x \in S$ is unknown if x is not bound to an existing object, true when x identifies an element of S, false otherwise.

The set containing all objects considered relevant for the discussion is called *universe of interest* or simply *universe* and denoted by $\mathcal{U}$. It is supposed that $\mathcal{U}$

contains as many constants as needed and all the objects that can be built from them by defining new sets. In particular, $\varnothing \in \mathcal{U}$, for every element $x \in \mathcal{U}$, $\{x\} \in \mathcal{U}$ and for every couple of sets $\{\alpha\} \in \mathcal{U}$ and $\{\beta\} \in \mathcal{U}$, $\{\alpha, \beta\} \in \mathcal{U}$ where α and β represent the content of the sets.

It is assumed that the reader is familiar with first order logic; the book of Mendelson [24] is taken as the primary source for the notation with some minor changes. The symbols $\mathtt{T}$ and $\mathtt{F}$ are used to denote the truth values *true* and *false*, respectively. The connectives *negation* $\neg\cdot$, *conjunction* $\cdot \wedge \cdot$, *inclusive disjunction* $\cdot \vee \cdot$, *implication* $\cdot \Rightarrow \cdot$ and *equivalence* $\cdot \Leftrightarrow \cdot$, together with the *universal quantifier* $\forall \cdot$ and *existential quantifier* $\exists \cdot$ with their usual meaning [24], are used to build new terms from existing ones. In particular, if α and β are terms, x a variable and S a set then $(x \in S)$, $(\neg\alpha)$, $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \Rightarrow \beta)$, $(\alpha \Leftrightarrow \beta)$, $((\forall x)\alpha)$ and $((\exists x)\alpha)$ are also terms. Parentheses are necessary to obtain unambiguous terms; however, they will be left implicit in most cases giving a different priority to each logic operator. For restoring the missing parentheses, operators have to be considered in this order: first all $\in$ not part of a quantification, then $\neg$, $\wedge$, $\vee$, $\forall$, $\exists$, $\Rightarrow$, $\Leftrightarrow$; connectives of the same kind from left to right, while consecutive negations and similar quantifications from right to left. The dot symbol $\,.\,$ is also adopted to mark the scope of a quantifier: in such case the scope extends from the dot to the first unmatched close parenthesis or the end of the term if such parenthesis does not exist. Furthermore, the abbreviation $((\forall x \in S)\alpha)$ is used instead of $((\forall x)((x \in S) \Rightarrow \alpha))$, similarly $(x \notin S)$ may replace terms like $(\neg(x \in S))$. Terms of the form $((x \in S) \wedge (y \in S))$ can be contracted in $(x, y \in S)$. Similarly, a sequence of quantifications of the same kind may be aggregated under a single symbol. For example, by applying the given conventions the term $(((\forall x)(x \in A) \Rightarrow ((\exists y)(y \in B) \Rightarrow (\alpha \vee (\neg\beta)))) \wedge ((\forall x)(x \in A) \Rightarrow \gamma))$ can be compactly denoted as $(\forall x \in A \,.\, \exists y \in B \,.\, \alpha \vee \neg\beta) \wedge \forall x \in A \,.\, \gamma$.

New sentences can be built on previously defined objects and new classes of objects can be declared by capturing their relevant properties with formal logic expressions. The *equivalence by definition* operator $\cdot \mathrel{\stackrel{\triangle}{\Longleftrightarrow}} \cdot$ enables the creation of a new set S enclosing it in a single sentence of the form $((\forall x \in \mathcal{U})((x \in S) \mathrel{\stackrel{\triangle}{\Longleftrightarrow}} \alpha))$, where x is used in α to state the properties of its elements. Parentheses can be omitted following the conventions given above, obtaining the generic term $\forall x \in \mathcal{U} \,.\, x \in S \mathrel{\stackrel{\triangle}{\Longleftrightarrow}} \alpha$ that can be further restated in the more compact notation $S \triangleq \{x \in \mathcal{U} \mid \alpha\}$ whenever necessary. The introduced notation $S \triangleq \{x \in \mathcal{U} \mid \alpha\}$ is very similar to an intensional definition but it is safe to use. For example, let us consider the Russell's paradox: in naive set theory one can build a set $R = \{x \mid x \notin x\}$ made of all sets that do not belongs to itself. If $R \in R$ then from the definition $R \notin R$ which is a contradiction. Conversely, if $R \notin R$ then $R \in R$ which again produces a contradiction. With the given notation, one can define $R \triangleq \{x \in \mathcal{U} \mid x \notin x\}$ from which it follows $\forall x \in \mathcal{U} \,.\, x \in R \Leftrightarrow x \notin x$ that in turn can be rewritten as $\forall x \,.\, x \notin \mathcal{U} \vee (x \in R \Leftrightarrow x \notin x)$. Assuming $R \in R$ it follows $R \notin \mathcal{U} \vee (R \in R \Leftrightarrow R \notin R)$; hence, $R \notin \mathcal{U}$ because the second external operand is always false. The same result holds starting from $R \notin R$: in any case no contradiction is reached.

The *containment* operator $\cdot \subseteq \cdot$ is used to state that all elements of a set A are contained into another set B; using the introduced notation this can be defined as $\forall A, B \in \mathcal{U} \,.\, A \subseteq B \mathrel{\stackrel{\triangle}{\Longleftrightarrow}} \forall x \in \mathcal{U} \,.\, x \in A \Rightarrow x \in B$. Similarly, the *equality* operator $\cdot = \cdot$

is used to denote a pair of sets with the same elements, namely $\forall A, B \in \mathcal{U} \,.\, A = B \triangleq A \subseteq B \wedge B \subseteq A$. Given two sets A and B the *cartesian product* operator $\cdot \times \cdot$ builds a new set of pairs such that the first element is contained in A while the second one is contained in B: $\forall A, B \in \mathcal{U} \,.\, A \times B \triangleq \{\langle x, y\rangle \in \mathcal{U} \mid x \in A \wedge y \in B\}$. Finally, the *power set* operator $\wp(\cdot)$ applied to a set A returns the set of all subsets of A, namely $\forall A \in \mathcal{U} \,.\, \wp(A) \triangleq \{P \in \mathcal{U} \mid P \subseteq A\}$.

A *relation* between two sets A and B is any subset of their cartesian product, i.e. $R \subseteq A \times B$. The set of elements that appear as the first element in a relation pair constitutes its *domain* and their are captured by the $dom(\cdot)$ operator as follows: $\forall R \subseteq A \times B \,.\, dom(R) \triangleq \{x \in \mathcal{U} \mid \langle x, y\rangle \in R\}$; while the set of elements that appear as the second element in a relation pair represents its *range* which is captured by the $range(\cdot)$ operator as: $\forall R \subseteq A \times B \,.\, range(R) \triangleq \{y \in \mathcal{U} \mid \langle x, y\rangle \in R\}$. A function f is a relation between two sets A and B with the additional constraint that for each element of its domain there exists one and only one corresponding element in its range: $\forall f \subseteq A \times B \,.\, \forall x \in A \,.\, \forall y, z \in B \,.\, \langle x, y\rangle \in f \wedge \langle x, z\rangle \in f \Rightarrow y = z$. Since there is a unique correspondence between an element $x \in A$ and the corresponding element $y \in B$ such that $\langle x, y\rangle \in f$, the element y is also denoted as $f(x)$. A function $f \subseteq A \times B$ is usually referred as $f : A \to B$, while $A \to B$ denotes the set of functions with domain A and range B. In the following the notation $\exists f : A \to B$ stands for exists a function f with domain A and range B. A function is said to be *partial* if it is not defined for all elements of its domain and it is denoted using the symbol $\rightharpoonup$, while the symbol $\uparrow$ stands for undefined.

The *power* of a finite set $A \subseteq \mathcal{U}$ is denoted by A^n and inductively defined as:

$$\forall n \in \mathbb{N} \cup \{0\} \,.\, A^n = \begin{cases} \varepsilon & \text{if } n = 0 \\ A & \text{if } n = 1 \\ A \times A^{n-1} & \text{otherwise} \end{cases}$$

Notice that all power sets are disjoint, that is $\forall n \in \mathbb{N} \cup \{0\},\ A^n \cap A^{n+1} = \varnothing$, this also implies the fundamental assumption that $\varepsilon \notin A$.

Similarly, the *closure* of a set $A \subseteq \mathcal{U}$, denoted as A^*, is defined as follows:

$$A^* = \bigcup_{n \in \mathbb{N} \cup \{0\}} A^n$$

An element $\sigma \in A^*$ is called *sequence* and it is represented as a list of A elements separated by commas and enclosed between angled brackets $\langle a_1, a_2, \ldots, a_n\rangle$, or more compactly as $\langle a_i\rangle_{i=1}^n$ for any $n \in \mathbb{N} \cup \{0\}$. For convention, whenever $n = 0$, $\langle a_i\rangle_{i=1}^0 = \langle\rangle = \varepsilon$. When is clear from the context, a sequence $\langle a_i\rangle_{i=1}^n$ can also be simply represented as list of elements $a_1 a_2 \ldots a_n$ one after the other.

The *elements* of a sequence $\sigma \in A^*$ are denoted by $set(\sigma)$, while a single indexed element a_i of a sequence $\sigma = \langle a_i\rangle_{i=1}^n$ is called *occurrence* of σ. The *inclusion* of an element $a \in A$ in a sequence $\sigma \in A^*$ is denoted as $a \in \sigma$. The length of a finite sequence $\sigma \in A^*$ is the number of occurrences in σ and is denoted as $|\sigma|$, namely $\forall \sigma \in A^* \,.\, |\sigma| = n \triangleq \exists n \in \mathbb{N} \cup \{0\} \,.\, \sigma \in A^n$.

The *concatenation* of two sequences $\alpha, \beta \in A^*$ is the sequence $\sigma \in A^*$ made of the symbols of α followed by the symbols of β respecting their order, it is denoted as $\alpha \circ \beta$, or simply as $\alpha\beta$ if it is clear from the context.

$$\begin{aligned}&\forall \alpha, \beta, \sigma \in A^* \,.\, \alpha \circ \beta = \sigma \triangleq \\ &\quad \alpha = \langle a_i \rangle_{i=1}^{n} \quad \wedge \quad \beta = \langle b_j \rangle_{j=1}^{m} \quad \wedge \quad \sigma = \langle s_k \rangle_{k=1}^{n+m} \quad \wedge \\ &\quad \forall k \in [1, n] \,.\, s_k = a_k \quad \wedge \quad \forall k \in [1, m] \,.\, s_{n+k} = b_k\end{aligned} \tag{2.1}$$

The tuple $(A^*, \circ, \varepsilon)$ is a monoid: the set is close under the concatenation of sequences, $\forall \alpha, \beta \in A^* \,.\, \alpha \circ \beta \in A^*$, the concatenation is associative, $\forall \alpha, \beta, \gamma \in A^* \,.\, (\alpha \circ \beta) \circ \gamma = \alpha \circ (\beta \circ \gamma)$, and there exists an identity element ε such that $\forall \sigma \in A^* \,.\, \varepsilon \circ \sigma = \sigma \circ \varepsilon = \sigma$.

The *restriction* or *projection* [25] of a sequence $\sigma \in A^*$ with respect to another set $B \subseteq A$ is a new sequence denoted as $\pi(B, \sigma)$ where all the occurrences not in B are removed.

$$\forall B \subseteq A \,.\, \forall \sigma \in A^* \,.\, \pi(B, \sigma) = \begin{cases} \varepsilon & \text{if } \sigma = \varepsilon \\ \pi(B, \gamma) & \text{if } \sigma = \gamma \circ \langle a \rangle \wedge a \notin B \\ \pi(B, \gamma) \circ \langle a \rangle & \text{if } \sigma = \gamma \circ \langle a \rangle \wedge a \in B \end{cases} \tag{2.2}$$

Clearly, $\pi(\varnothing, \sigma) = \varepsilon$ and $\pi(A, \sigma) = \sigma$. The projection can be extended to a set of sequences in the following way: $\forall S \subseteq A^*$, $\pi(A, S) = \{\pi(A, \sigma) \mid \sigma \in S\}$.

The *number of occurrences* of a set of symbols $B \subseteq A$ in a sequence $\sigma \in A^*$ is denoted by $|\sigma|_B = |\pi(B, \sigma)|$. When B is the singleton $\{a\}$, the notation $|\sigma|_a$ is used instead of $|\sigma|_{\{a\}}$.

The *prefixes* [25] of a sequence $\sigma \in A^*$ is denoted with $\mathit{prefixes}(\sigma)$ defined as:

$$\mathit{prefixes}(\sigma) = \{\alpha \in A^* \mid \exists \beta \in A^* \,.\, \sigma = \alpha \circ \beta\} \tag{2.3}$$

This operation can be extended to set of sequences considering the union of the prefixes of all these sequences: $\forall S \subseteq A^*$, $\mathit{prefixes}(S) = \bigcup_{\sigma \in S} \mathit{prefixes}(\sigma)$.

The sequence notions given above can be applied in the language context using a different jargon: the finite set of symbols $\Sigma \subseteq \mathcal{U}$ is called *alphabet*, any subset of the closure of Σ, $L \subseteq \Sigma^*$ is said to be a *language* L over Σ, and each finite sequence $\sigma \in \Sigma^*$ is said to be a *string*.

Structures

The following chapters make heavy use of mathematical structures in order to formalize the expressed concepts. Mathematical structures are usually defined in terms of tuples and operations on them; for instance, a graph is usually stated as a pair $G = \langle V, E \rangle$ such that $E \subseteq V \times V$, where the chosen symbols G, V, and E are not relevant for the purpose of defining the structure. When two or more structures of the same type are needed, it is a common practice to use subscripts and/or superscripts for distinguishing them. For example, one may define the union of two graphs $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$, as $G' = \langle V', E' \rangle$ such that $V' = V_1 \cup V_2$ and $E' = E_1 \cup E_2$. This mathematical notation can soon become cumbersome, especially when one has to simultaneously deal with several different structures having components with the same name.

In these chapters a different approach is adopted: a *structure* $\mathcal{S}$ is defined as a set of objects, that in turn are tuples of the form $\langle X_i \rangle_{i=1}^{n}$, having the same properties. Each component X_i of a tuple $\langle X_i \rangle_{i=1}^{n} \in \mathcal{S}$ is denoted by a different access function with signature $f_i : \mathcal{S} \to \wp(\mathcal{C}_i)$, called *field* that is nothing more than a total function with domain the structure $\mathcal{S}$ and range the parts of the domain $\mathcal{C}_i$ of its corresponding component $X_i \subseteq \mathcal{C}_i$. The only drawback is that for every structure $\mathcal{S}$ there exist n fields $f_i : \mathcal{S} \to \mathcal{C}_i$, such that for every object $s \in \mathcal{S}$ of the form $s = \langle X_i \rangle_{i=1}^{n}$, $f_i(s) = X_i$. The set of all fields $\{f_i\}_{i=1}^{n}$ of a structure $\mathcal{S}$ is denoted by $\mathit{fields}(\mathcal{S})$. For instance, the structure of a graph can be defined as $\mathcal{G} = \{(V, E) \mid E \subseteq V \times V\}$ where $\mathit{fields}(\mathcal{G}) = \{\mathit{vertices}, \mathit{edges}\}$. In particular, for any $g \in \mathcal{G}$ the access functions $\mathit{vertices} : \mathcal{G} \to \wp(\mathcal{U})$ and $\mathit{edges} : \mathcal{G} \to \wp(\mathcal{U})$ denote the vertices and the edges of g, respectively. The signature of such functions can be improved assuming that all vertices are chosen from a generic set $\mathcal{V} \subseteq \mathcal{U}$, hence fields can be restated as $\mathit{vertices} : \mathcal{G} \to \wp(\mathcal{V})$, and $\mathit{edges} : \mathcal{G} \to \wp(\mathcal{V} \times \mathcal{V})$, making the structure at hand even more explicit.

The union of two graphs $g, h \in \mathcal{G}$ can then be defined as the graph $u \in \mathcal{G}$ such that $\mathit{vertices}(u) = \mathit{vertices}(g) \cup \mathit{vertices}(h)$, and $\mathit{edges}(u) = \mathit{edges}(g) \cup \mathit{edges}(h)$. In this way, no subscripts or superscripts are needed; furthermore, different structures can have access functions with the same name or symbol, since they can be distinguished by their domain. For instance, the union of graphs mentioned above may be stated as a total function $\mathit{union} : \mathcal{G} \times \mathcal{G} \to \mathcal{G}$ and denoted by $\cdot \cup \cdot$. This notation should not surprise a reader familiar with general-purpose functional languages.

Pattern matching is used to retrieve the attributes of a tuple when considered as a single unit. For example, given a structure $\mathcal{S}$ with two fields, the notation $\exists u, v \in \mathcal{U} \,.\, \langle u, v \rangle = s$ can be used to extract the attributes of any object $s \in \mathcal{S}$. If an attribute is not relevant for the context, it can be substituted by the underscore symbol "_" that is a placeholder for any fresh variable not referenced in the context, i.e. $\exists v \in \mathcal{U} \,.\, \langle \_, v \rangle = s$ stands for $\exists u \in \mathcal{U} \,.\, \exists v \in \mathcal{U} \,.\, a = \langle u, v \rangle$.

The remainder of this chapter introduces the multiset and the graph structures in great details. The aim is twofold: to show how structures are declared in practice, and to describe two basic structures extensively used in the following chapters. The reader can safely skip this part without loss of continuity.

Multiset

A multiset is usually defined as a total function $m : \mathcal{U} \to \mathbb{N} \cup \{0\}$ such that for each element of the domain it returns the number of its occurrences. For demonstration purposes, the same effect is obtained by considering functions defined over finite objects, as in the following definition.

Definition 2.1 (Multiset). A *multiset* is a partial function $f : \mathcal{U} \rightharpoonup \mathbb{N}$ over a finite domain, namely $f \subseteq \mathcal{U} \times \mathbb{N}$ is a multiset if and only if $|dom(f)| < \omega$, and for all $(u, i), (v, j) \in f$ it holds that $u = v$ implies $i = j$. The set of all possible multisets is denoted by $\mathcal{M}$ defined as:

$$\mathcal{M} \triangleq \{\langle f \rangle \mid f : \mathcal{U} \rightharpoonup \mathbb{N} \wedge |dom(f)| < \omega\} \tag{2.4}$$

Tab. 2.1 summarizes the main functions over the structure $\mathcal{M}$. In particular, this structure is so simple to have only one component and consequently only one field $pairs : \mathcal{M} \to \wp(\mathcal{U} \times \mathbb{N})$ returning the internal representation of each multiset as a finite set of pairs. The fields declaration becomes $fields(\mathcal{M}) \triangleq \{pairs\}$. The first

Table 2.1. Basic operations on multisets.

Symbol	Function	Description	Ref
	$pairs : \mathcal{M} \to \wp(\mathcal{U} \times \mathbb{N})$	The internal finite set of pairs representing the multiset.	*inline*
	$elements : \mathcal{M} \to \wp(\mathcal{U})$	Distinct elements contained in the multiset.	*inline*
μ	$multiplicity : \mathcal{M} \times \mathcal{U} \to \mathbb{N} \cup \{0\}$	Multiplicity of a particular element.	*inline*
	$multiset : \mathcal{U} \to \mathcal{M}$	Multiset constructor.	*inline*
$\cdot \in \cdot$	$in : \mathcal{M} \times \mathcal{U} \to \mathbb{B}$	True if the element belongs to the multiset, false otherwise.	*inline*
$\lvert \cdot \rvert$	$size : \mathcal{M} \to \mathbb{N} \cup \{0\}$	Total number of elements in the multiset.	Eq. 2.5
$\cdot \cup \cdot$	$union : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$	Union between two multisets.	Eq. 2.6
$\cdot \setminus \cdot$	$subtract : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$	Difference between two multisets.	Eq. 2.8

column of the table contains an optional symbol that can be used to compactly denote the function, the second column contains the function signature, the third column contains a brief description of the function behaviour, and the last column contains a reference to the expression that formally define the function. This last column can contain the label *inline* which means that the function is defined inline inside the text, rather than on a specific numbered equation. Fields are usually declared inline, because they have a straightforward definition.

For any multiset $m \in \mathcal{M}$, the function $elements(m) = \{u \in \mathcal{U} \mid \exists i \in \mathbb{N} \,.\, (u, i) \in pairs(m)\}$ returns the set of elements contained in the multiset. The function $multiplicity : \mathcal{M} \times \mathcal{U} \to \mathbb{N} \cup \{0\}$ returns the multiplicity of each element u in the

multiset m, formally $multiplicity(m,u) = i$ if $\exists i \in \mathbb{N} \,.\, (u,i) \in pairs(m)$, 0 otherwise. The function $in : \mathcal{M} \times \mathcal{U} \to \mathbb{B}$ determines if an element u belongs to the multiset m, formally $in(m,u) = \mathtt{T}$ if $u \in elements(x)$, $\mathtt{F}$ otherwise. It worths noting that $\forall u \in \mathcal{U} \,.\, in(m,u) = \mathtt{T} \Leftrightarrow \mu(m,u) > 0$. The function $multiset : \mathcal{U} \to \mathcal{M}$ builds a multiset starting from a single element $u \in \mathcal{U}$, formally $\forall u \in \mathcal{U} \,.\, \forall m \in \mathcal{M} \,.\, m = multiset(u) \overset{\triangle}{\Longleftrightarrow} pairs(m) = \{(u,1)\}$.

The function $size : \mathcal{M} \to \mathbb{N} \cup \{0\}$ returns the total number of elements in the multiset and is defined in Eq. 2.5.

$$\begin{aligned} &size : \mathcal{M} \to \mathbb{N} \cup \{0\} \ \ is \\ &\quad \forall m \in \mathcal{M} \,.\, size(m) = \sum_{(u,i) \in pairs(m)} i \end{aligned} \tag{2.5}$$

The $union : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$ operator returns a new multiset obtained from the union of two other multisets, its definition is reported in Eq. 2.6.

$$\begin{aligned} &union : \mathcal{M} \times \mathcal{M} \to \mathcal{M} \ \ is \\ &\quad \forall x, y, z \in \mathcal{M} \ . \ z = union(x,y) \overset{\triangle}{\Longleftrightarrow} \\ &\qquad pairs(z) = \{\langle u, i \rangle \subseteq \mathcal{U} \times \mathbb{N} \mid u \in elements(x) \cup elements(y) \wedge \\ &\qquad\qquad i = \mu(x,u) + \mu(y,u)\} \end{aligned} \tag{2.6}$$

Notice that functions are defined on fields in order to decouple them from the internal structure. Nevertheless, a function can be defined in more compact way if fields can be inferred from the definition. For instance, the field *pairs* can be defined as $pairs(z) = \{\langle u, \mu(u) \rangle \mid u \in \mathcal{U} \wedge \mu(u) > 0\}$, in this case the function *union* can be rewritten as follows:

$$\begin{aligned} &union : \mathcal{M} \times \mathcal{M} \to \mathcal{M} \ \ is \\ &\quad \forall x, y, z \in \mathcal{M} \ . \ z = union(x,y) \overset{\triangle}{\Longleftrightarrow} \\ &\qquad \forall u \in \mathcal{U} \,.\, \mu(z,u) = \mu(x,u) + \mu(y,u) \end{aligned} \tag{2.7}$$

Given the constructor function $multiset : \mathcal{U} \to \mathcal{M}$, the addition of a single element $u \in \mathcal{U}$ to a multiset $m \in \mathcal{M}$ is compactly denoted as $m \cup u$, which stands for $m \cup multiset(u)$.

Function $subtract : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$ returns a new multiset equal to the difference between two given multisets, as defined in Eq. 2.8.

$$\begin{aligned} &subtract : \mathcal{M} \times \mathcal{M} \to \mathcal{M} \ \ is \\ &\quad \forall x, y, z \in \mathcal{M} \,.\, z = subtract(x,y) \overset{\triangle}{\Longleftrightarrow} \\ &\qquad \forall u \in \mathcal{U} \,.\, \mu(z,u) = \mu(x,u) \ominus \mu(y,u) \end{aligned} \tag{2.8}$$

where $x \ominus y$ is the substraction operation defined on natural numbers.

Similarly to the previous operation, the subtraction of a single element $u \in \mathcal{U}$ from a multiset $m \in \mathcal{M}$ is briefly denoted as $m \setminus u$, which stands for $m \setminus multiset(u)$.

A multiset $m \in \mathcal{M}$ can be extensionally defined by declaring its internal pairs $m \triangleq \langle \{e_i \mapsto v_i\}_{i=1}^{n} \rangle$, or with an abuse of notation as $m \triangleq \{e_i \mapsto v_i\}_{i=1}^{n}$, where $e_i \in \mathcal{U}$ is an element of m, v_i is its multiplicity, and n is the number of distinct elements in m. A different and more compact notation for multiset is defined here, where the two elements of each pair are putted together and separated by a reversed superscript $\backprime$, namely $m = \{v_i \backprime e_i\}_{i=1}^{n}$.

Graph

The multiset definition and its functions should be intended as a basic example, since a multiset as only one independent field. This section shows how to deal with multiple fields and how a new structure can be defined by difference from an existing one.

Definition 2.2 (Directed Graph). A *directed graph* G is a tuple $\langle V, E \rangle$ that belongs to the set $\mathcal{G}$ defined as follows

$$\mathcal{G} \triangleq \{\langle V, E \rangle \mid V \subseteq \mathcal{U} \ \wedge E \subseteq V \times V\} \tag{2.9}$$

The basic operations on graphs are summarized in Tab. 2.2. In this case the structure has two fields $\mathit{fields}(\mathcal{G}) \triangleq \{\mathit{vertices}, \mathit{edges}\}$ with signature $\mathit{vertices} : \mathcal{G} \to \wp(\mathcal{U})$, and $\mathit{edges} : \mathcal{G} \to \wp(\mathcal{U} \times \mathcal{U})$, one for accessing vertices and one for edges, respectively. For any graph $g \in \mathcal{G}$ such that $g = \langle V, E \rangle$, $\mathit{vertices}(g) = V$, and $\mathit{edges}(g) = E$.

Table 2.2. Basic operations on graphs.

Symbol	Function	Description	Ref
	$\mathit{vertices} : \mathcal{G} \to \wp(\mathcal{U})$	Vertices of the graph.	*inline*
	$\mathit{edges} : \mathcal{G} \to \wp(\mathcal{U} \times \mathcal{U})$	Edges of the graph.	*inline*
	$\mathit{predecessors} : \mathcal{G} \times \mathcal{U} \rightharpoonup \wp(\mathcal{U})$	Set of predecessors of a vertex.	*inline*
	$\mathit{successors} : \mathcal{G} \times \mathcal{U} \rightharpoonup \wp(\mathcal{U})$	Set of successors of a vertex.	*inline*
	$\mathit{is\text{-}predecessor} : \mathcal{G} \times \mathcal{U} \times \mathcal{U} \rightharpoonup \mathbb{B}$	True if the first vertex is a predecessor of the second one.	*inline*
	$\mathit{is\text{-}successor} : \mathcal{G} \times \mathcal{U} \times \mathcal{U} \rightharpoonup \mathbb{B}$	True if the first vertex is a successor of the second one.	*inline*
$\cdot \cup \cdot$	$\mathit{union} : \mathcal{G} \times \mathcal{G} \to \mathcal{G}$	Compute the union of two graphs.	Eq. 2.10
$\cdot \bowtie \cdot$	$\mathit{join} : \mathcal{G} \times \mathcal{G} \to \mathcal{G}$	Compute the join of two graphs.	Eq. 2.11

The function $\mathit{predecessors} : \mathcal{G} \times \mathcal{U} \rightharpoonup \wp(\mathcal{U})$ returns the set of vertices that are predecessors of the given vertex, formally $\forall g \in \mathcal{G} \,.\, \forall u \in \mathit{vertices}(g) \,.\, \mathit{predecessors}(g, u) = \{v \in \mathcal{U} \mid (v, u) \in \mathit{edges}(g)\}$, undefined otherwise. Similarly, the $\mathit{successors} : \mathcal{G} \times \mathcal{U} \rightharpoonup \wp(\mathcal{U})$ function returns the set of vertices that are successors of the given vertex, hence $\forall g \in \mathcal{G} \,.\, \forall u \in \mathit{vertices}(g) \,.\, \mathit{successors}(g, u) = \{v \in \mathcal{U} \mid (u, v) \in \mathit{edges}(g)\}$, undefined otherwise. The function $\mathit{is\text{-}predecessor} : \mathcal{G} \times \mathcal{U} \times \mathcal{U} \to \mathbb{B}$ returns true if the first given vertex is a predecessor of the second one, formally $\forall g \in \mathcal{G} \,.\, \forall u, v \in$

$vertices(g)$. $is\text{-}predecessor(g, u, v)$ = T if $v \in predecessors(g, u)$, F otherwise. Similarly, the function $is\text{-}successor : \mathcal{G} \times \mathcal{U} \times \mathcal{U} \to \mathbb{B}$ returns true if the first given vertex is a successor of the second one, hence $\forall g \in \mathcal{G} \,.\, \forall u, v \in vertices(g) \,.\, is\text{-}successor(g, u, v) =$ T if $v \in successors(g, u)$, F otherwise. Both functions are undefined if one of the first two given arguments is not a vertex of g.

The $union : \mathcal{G} \times \mathcal{G} \to \mathcal{G}$ operator performs the union between two graphs: in particular, the resulting graph will contain the union of the original vertices and edges. It is formally defined in Eq. 2.10.

$$\begin{aligned} &union : \mathcal{G} \times \mathcal{G} \to \mathcal{G} \;\; is \\ &\quad \forall g, h, d \in \mathcal{G} \,.\, d = union(g, h) \overset{\triangle}{\iff} \\ &\qquad vertices(d) = vertices(g) \cup vertices(h) \wedge \\ &\qquad edges(d) = edges(g) \cup edges(h) \end{aligned} \tag{2.10}$$

Finally, the $join : \mathcal{G} \times \mathcal{G} \to \mathcal{G}$ operator given in Eq. 2.11 is similar to the union one, but in this case an extra edge is created between each pair of vertices that originally belong to a different a graph.

$$\begin{aligned} &join : \mathcal{G} \times \mathcal{G} \to \mathcal{G} \\ &\quad \forall g, h, d \in \mathcal{G} \,.\, d = join(g, h) \overset{\triangle}{\iff} \\ &\qquad vertices(d) = vertices(g) \cup vertices(h) \wedge \\ &\qquad edges(d) = edges(g) \cup edges(h) \cup \\ &\qquad\qquad \{(u, v) \mid u \in vertices(g) \wedge v \in vertices(h)\} \end{aligned} \tag{2.11}$$

A function over a structure often returns a new structure similar to the original one; hence, it may be useful to define a new structure by difference, assuming that certain fields do not change. For instance, let us consider the function $add\text{-}edge : \mathcal{G} \times \mathcal{U} \to \mathcal{G}$, which returns a new graph equal to the original one except for an additional edge, it can be defined as follows:

$$\begin{aligned} &add\text{-}edge : \mathcal{G} \times \mathcal{U} \to \mathcal{G} \;\; is \\ &\quad \forall g, h \in \mathcal{G} \,.\, \forall u, v \in vertices(g) \,. \\ &\qquad h = add\text{-}edge(g, u, v) \overset{\triangle}{\iff} \\ &\qquad\quad \forall f \in fields(\mathcal{G}) \setminus \{edges\} \,.\, f(h) = f(g) \wedge \\ &\qquad\quad edges(h) = edges(g) \cup \{(u, v)\} \end{aligned} \tag{2.12}$$

The same definition can be obtained using:

$$\begin{aligned} &add\text{-}edge : \mathcal{G} \times \mathcal{U} \to \mathcal{G} \;\; is \\ &\quad \forall g, h \in \mathcal{G} \,.\, \forall u, v \in vertices(g) \,. \\ &\qquad h = add\text{-}edge(g, u, v) \overset{\triangle}{\iff} \\ &\qquad\quad h \;\; same \;\; as \;\; g \;\; except \\ &\qquad\quad edges(h) = edges(g) \cup \{(u, v)\} \end{aligned} \tag{2.13}$$

where "x *same as* y *except*" means that all fields are equal in the two structures except for those listed below.

3

Graphical Process Modeling Languages

Plenty of graphical PMLs have been proposed over the years: a complete analysis concerning all of them cannot fit in a single chapter and is out the scope of this thesis. Conversely, this chapter focuses on some well-known graphical modeling languages that are valid representatives for the category. In particular, it provides an overview of Petri Nets, Coloured Petri Nets (CPNs), Yet Another Workflow Language (YAWL), and Business Process Model and Notation (BPMN). Petri nets has been chosen because it is the most simple formal language able to model many interesting aspects of concurrent systems in a graphical way. Furthermore, it represents the theoretical foundation of many other modeling languages used in the PAIS domain, including the ones considered here. Petri nets theory is also used through the whole thesis and in particular in the next chapter. Several variants of Petri nets have been proposed in literature; for this reason, a general language called Place Transition Nets (PTNs) is introduced here, that captures many variants of interest in a complete and formal way. CPNs are a high-level variant of Petri nets that is treated in a separate section, because it integrates a functional language for representing and manipulating data. YAWL is both a modeling language routed on Petri nets and a fully-fledged workflow system. It was born as an academic research project to show how a PML can support many of the existing workflow control-flow patterns in a uniform way. BPMN is an Object Management Group (OMG) industrial standard adopted by many offerings, including GlassFish Application Server [26] and JBoss jBPM Workflow Engine [27]. The set of constructs offered by the BPMN is quite large; hence, this chapter concentrates only on the basic ones that have a clearly stated semantics. Each language is presented in a similar way starting from the description of its graphical elements, together with its syntax and an intuitive interpretation, then a formalization is provided in terms of mathematical structures.

The aim is to give a sufficiently wide overview of the state of the art about PMLs by providing a clear and uniform treatment of the four considered languages.

The remainder of this chapter is organized as follows: Sec. 3.1 discusses some related work about PMLs classification and about other modeling languages not considered here. Sec. 3.2 introduces the PTNs language from which many Petri nets variants can be derived by adding syntactical restrictions. Sec. 3.3 is dedicated to CPNs. YAWL is discussed in Sec. 3.4, while BPMN is treated in Sec. 3.5.

3.1 Related Work

The workflow pattern initiative [28] was born with the aim to evaluate the suitability of PMLs and workflow systems in terms of a set of recurring features, called patterns. Many kinds of patterns have been defined, the principal ones cover three different aspects: control-flow [29], data [30] and resources [31]. Several PMLs and workflow systems have been evaluated using this methodology [32–35], providing a useful starting point for their comparison.

In [36] Börger performs a critical analysis about BPMN, YAWL and workflow patterns, claiming that they fail in realizing their original vision. In particular, to be a standard, BPMN contains too much ambiguities in the original description resulting in an underspecification of semantically relevant concepts. Furthermore, the large number of offered constructs may aggravate this problem. Finally, the main critics moved by Börger about YAWL is its strict dependency to the workflow control-flow patterns and the lack of a completely formalized semantics.

Scientific Workflow Management Systems (WfMSs) are software systems developed for automating large-scale scientific experiments. Their main goals are reusing domain specific functions and tools, and ease their integration through a visual editor. Moreover, the nature of the defined computations may require cluster of computers and remote resources; therefore, many scientific WfMSs provides a support for transparently executing their tasks on a Grid environment. Some representative scientific WfMSs are Kepler [37], Taverna [38] and Triana [39], they have been analyzed and compared with respect to traditional WfMSs using workflow patterns in [35]. The presence of different systems is justified by their different application domains: each system provides a set of reusable components for a specific science context, such as biology, chemicals, physics, and so on. They are not treated here for several reasons. First of all, despite the rich set of components offered for a specific application domain, the provided routing constructs are limited and they can be simulated through CPNs structures [35]. Moreover, their main aim is the automation of complex computations, rather than the coordination of human agents. Finally, they are usually offered as stand-alone applications without a client/server architecture.

The aim of the WIDE project (Workflow on Intelligent Distributed database Environment) [40,41] is to extends the technology underlying traditional DataBase Management Systems (DBMSs) in order to support process-oriented applications. The main interesting contributions of such project are: the introduction of the business transaction notion, the support for exceptions, and the management of temporal aspects. A business transaction is an extension of the traditional transaction notion developed in the database context to the workflow domain. It is based on two main concepts: atomicity and isolation. Atomicity regards the identification of some process parts that have to be atomically executed, i.e. their execution can only reach the end or be unrolled. Isolation implies that intermediate results computed by some parts are hidden to the rest of the workflow and to the external environment. The exception handling mechanism is based on the Event-Condition-Action (ECA) paradigm: the event identifies when the exception has been thrown, the condition is an expression that determines if the exception has to be handled, and the action is the activity to be performed. Furthermore, each data item may

have a temporal aspect related to it that becomes of particular importance during exception handling. The WIDE model includes the definition of temporal instants, temporal intervals, and durations.

A different design paradigm known as object-aware process management is adopted in some PMLs. This approach gives great attention to the business objects manipulated during the process execution. More specifically, processes are described in terms of object behaviors and object interactions. The behavior of an object instance is captured by specific states and the transitions among them. The execution of an activity depends on the current state of an object and determines the activation of a subsequent state. Several approaches based on the object life cycle (OLC) notion have been defined in literature, for instance: object-centric process models [42], business artifacts [10], data-driven process coordination [43], and object-process methodology [44]. These approaches typically associate to each activity as set of pre/post-conditions related to object states. They usually cover process modeling but do not provide a direct support for process execution. PHILharmonicFlows [45] is a complete framework for object-aware process management. It supports the definition of data and processes in separated, but well-integrated models by explicitly considering the relationships between them. Furthermore, it provides support for process modeling, execution and monitoring.

The ADEPT$_{flex}$ system [46], recently renamed as AristaFlow BPM Suite, has been developed in the context of the ADEPT project [47] with the aim to improve the flexibility of existing PAISs. Such system is able to support the process schema evolution, namely the migration of process instances to a new schema version. This ability is of paramount importance in contexts characterized by long evolving processes or in which deviations from the standard procedures happens frequently, such as in the heath-care domain. The underlying rationale is based on the definition of a comprehensive set of changing operations with pre/post-conditions which ensure that, if the desired changes satisfies the preconditions, the resulting process schema will again be correct. A process described with AristaFlow is characterized by a control-flow described as a direct structured graph, namely a symmetric correspondence between split and join nodes is required, and data represented as global variables. Moreover, tasks in different branches can be synchronized through synchronization edges, e.g. delay dependencies. The use of structured forms speedups the analysis required to perform a particular change by restricting the area to be analyzed, and it simplifies the resulting structural adaptations.

3.2 Place Transition Nets

> "*We do not consider our list of important aspects of Petri nets complete, and for each aspect claimed to be common to all Petri net variants there might exist very reasonable exceptions.*"
> – Desel and Juhás, What Is a Petri Net? [48]

The original concept of *Petri nets* appears for the first time in the seminal work of Carl Adam Petri, where the author proposes a novel approach to the theory of computation that better matches the nature of physical information flow [49]. Nowadays, Petri nets can be better understood as a family of formal languages for modeling a wide range of aspects concerning concurrent systems: several Petri nets variants have been proposed, each one with its own set of features and analysis methods, tailored for a specific research or application field.

Nevertheless, these languages share some common characteristics and they are all driven by the same basic principles [50]. First of all, they have a graphical notation directly mapped to a formal semantics stated in terms of mathematical functions. They universally embody four principal language constructs, called here *place*, *transition*, *arc* and *token*. A place is usually depicted as an ellipse, a transition as a rectangle, an arc as a curved line ending with a solid arrow and a token as a small solid circle. An arc connects a place with a transition and vice versa but no arc can connect two elements of the same type, while tokens are ideally stored in places. A *net* is composed of such elements together with further textual inscriptions such as *identifiers*, *weights* and *capacities*. Petri nets elements can be interpreted in several ways depending on the underlying application domain [14]; for instance, they can be considered conditions, channels, counters, events, actions, tasks, messages, threads and so on.

The behavior of a transition is determined by its *locality* that includes itself and the places directly connected to it [50]. Transitions with disjoint localities are guaranteed to be independent, hence they can occur concurrently. Depending on the assumed semantics, this principle does not exclude either concurrent transitions with overlapping localities or concurrent instances of the same transition.

The proliferation of Petri nets variants is triggered by the compromise between language expressiveness and the power of the related algorithmic solutions: greater expressiveness enlarges the class of systems that can be captured, but at the same time precludes the existence of general verification methods and hinders the effectiveness of the existing ones.

The drawback of such richness is an undesired complexity of the Petri nets theory in comparison to the intuitive semantics of nets, that can be easily described in terms of tokens flowing from one place to the other through arcs by means of local transitions. Ironically, Petri nets graphical notation may be ambiguous due to the existence of many formal semantics [51,52]: the same net can be interpreted in several ways; hence, the graphical representation of a model should be always accompanied with formal definitions fixing the details about its semantics.

Several unification frameworks have been proposed to relate existing Petri nets variants [53,54]: in particular, a large group of Petri nets can be seen as extensions or restrictions of other ones. Extensions and restrictions that do not alter the class of representable nets are usually introduced to capture some concepts in a more

convenient way or to ease the verification of some desired properties. Conversely, when the added features increase or reduce language expressiveness, i.e. the kinds of representable nets, the proposed changes is said to be a *proper extension* or a *proper restriction*, respectively.

To expose the full potentials of Petri nets in the process modeling context, this section introduces them following the approach of Valk [55, 56] and Dufourd et al. [57]: it initially describes a reasonably general yet formal language, called here *place transition nets*, from which other Petri nets variants can be obtained by imposing further syntactical and semantical restrictions. Such language is said to be *reasonably* general, because there exist even more general extensions and other variants that cannot be captured in this way. However, this cannot be considered a great limitation, since the primary purpose of such extensions if far beyond the modeling of processes. Place/Transition Systems are the most used Petri Nets variants both from theoretical and practical perspectives [58]. This success is probably due to their expressiveness and their simpler *interleaved* or *sequential semantics* [51] where a transition is considered a single atomic change. Other Petri nets variants such as Condition/Event Systems and Elementary Net Systems have a so called *step semantics* [51, 58] where a step is a set of transitions that are all enabled at the same time and occur together. Petri nets languages with a step semantics are not treated further because they do not reflect the actual theoretical basis of PMLs. Coloured Petri Nets are also excluded from this section because will be discussed in Sec. 3.3.

3.2.1 Graphical Elements and Syntax

The graphical elements of PTNs together with their inscriptions are summarized in the first column of Fig. 3.1 from (a) to (m); the remaining two columns depict alternative representations of the same elements with some simplifying conventions. Neither the size of graphical elements nor the position of the inscriptions is relevant, provided that all inscriptions are close to their related constructs.

A place is depicted as an ellipse with a very close inscription of the form p/k as in Fig. 3.1.a. The inscription includes a mandatory unique identifier p followed by an optional number $k \in \mathbb{N}$, called the *capacity* of the place. When present, the capacity is separated by the place identifier with a slash symbol. As depicted in Fig. 3.1.b, a place can be enhanced with a further custom label ℓ, but such label is only an aid for the end-user: its presence does not alter the behaviour of a place. If the capacity is not specified as in Fig. 3.1.c it is assumed to be unlimited, i.e. $k = \omega$, where $\forall n \in \mathbb{N} \,.\, \omega > n$.

A transition is depicted as a rectangle with a very close inscription $t : \ell$, as exemplified in Fig. 3.1.d. The inscription includes a mandatory unique identifier t followed by an optional *label* ℓ, preceded by a colon. The label ℓ can also be placed inside the rectangle as in Fig. 3.1.e. A label can have different meaning depending on the application domain, but it is usually interpreted as an *action* that can be performed. An action does not need to be unique and when it is not specified as in Fig. 3.1.f, it is assumed to be the special symbol τ, called *silent action.*

An arc is depicted as in Fig. 3.1.g: a straight or curved line with a solid arrow at one end and a centered inscription of the form $w \cdot \hat{p}$ called *instantaneous weight*,

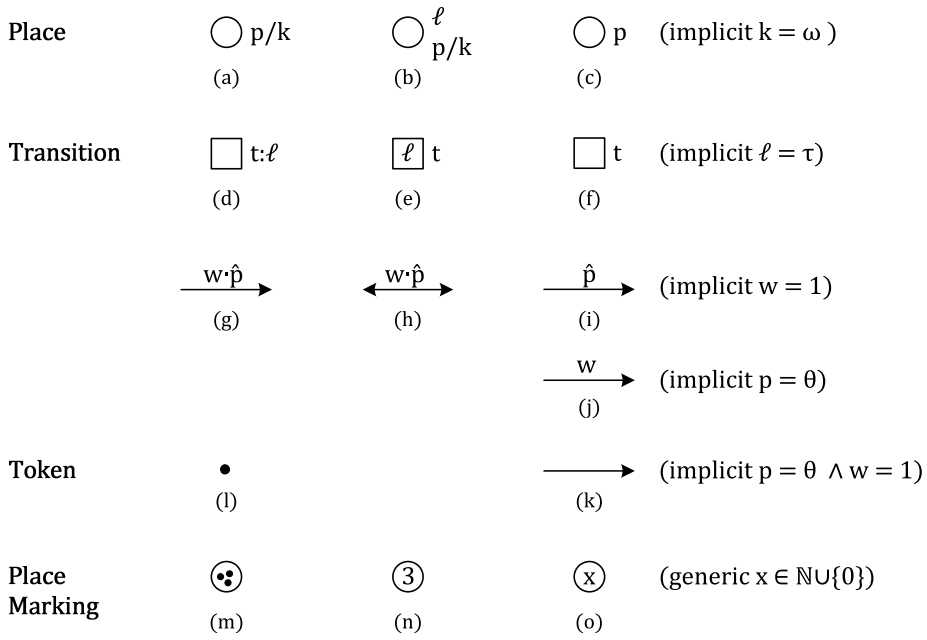


Fig. 3.1. PTNs graphical elements. (a) Place with explicit capacity. (b) Place with an additional label. (c) Place with implicit unlimited capacity. (d) Transition with explicit label. (e) Transition with explicit internal label. (f) Silent transition. (g) Generic arc of variable weight. (h) Generic double arc of variable weight. (i) Arc with implicit unitary weight. (j) Arc of constant weight w. (k) Simple arc. (l) Representation of a token. (m) A place marked with 3 tokens. (n) An alternative place marking. (o) A place marked with a generic number of tokens, possibly zero.

where $w \in \mathbb{N}$ is the *weight* and $\hat{p}$ is the *place reference* of the arc, i.e. the identifier of an existing place. Two arcs with the same ends and inscription but different orientation can be represented as a single arc with an arrow in both ends, as in Fig. 3.1.h. Neither the weight nor the place reference are mandatory: when the weight w is omitted, as in Fig. 3.1.i, it is assumed to be 1; when it is the place reference $\hat{p}$ to be omitted, as happens in Fig. 3.1.j, it is assumed to be the ideal place θ: a place that always contains one and only one token, in short $\hat{\theta} = 1$. An arc like the one in Fig. 3.1.k with weight $w = 1$ and place reference $\hat{p} = \hat{\theta}$ is said to be a *simple arc*.

A token is depicted as a small solid circle as in Fig. 3.1.l. One or more tokens are used to mark a place, e.g. the place in Fig. 3.1.m is marked with 3 tokens; it is also common to say that a marked place *contains* one or more tokens. An alternative representation is given in Fig. 3.1.n where tokens are replaced with their count: this becomes necessary when the number of tokens is too large to fit in the place representation. When the number of tokens of a place is not known, the place is marked with the name of a variable as happens in Fig. 3.1.o.

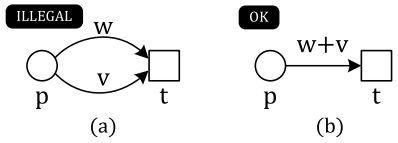


Fig. 3.2. (a) Multiple arcs with the same source and target are not supported by PTNs. (b) A weighted arc can represent multiple weighted arcs having the same orientation.

The PTNs syntax is pretty straightforward consisting of very few rules. A *net* is a graph made of a finite number of nodes eventually connected with arcs. A valid net shall comply with two simple syntactical rules: (1) an arc shall not connect two nodes of the same kind and (2) two nodes shall not be connected by more than one arc having the same orientation, i.e. the underlying graph is not a multi-graph. While the former is an essential rule shared by all Petri nets, the latter may be limiting in certain contexts, e.g. when the Petri nets language supports only constant arcs of unitary weight. Nevertheless, this is not a great limit in PTNs thanks to weighted arcs and silent transitions. For instance, two constant weighted arcs having the same source, target and orientation as in Fig. 3.2.a can be modeled with a single arc of a constant weight equals to the sum of the previous weights, as exemplified in Fig. 3.2.b. Notice that the white on black labels in Fig. 3.2 are not part of the introduced language, they are used to specify additional information about models.

In PTNs any specified arc shall be always connected to a source and a target node, while a node is not required to be connected to another one through an arc, i.e. the underlying graph may be not connected.

3.2.2 Bare Language Interpretation

The informal semantics of the PTNs language can be easily explained in terms of states and transitions. The *state* of a net is given by the distribution of tokens inside its places. In Petri nets literature, a state is frequently called *marking*, a term better reflecting its graphical nature. Chosen the place identifiers p_1 through p_m, a marking can be compactly expressed in polynomial-like form as $\langle \sum_{i=1}^{m} c_i p_i \rangle$ where c_i is the number of tokens in the place p_i and any component $c_i p_i$ of the sum is omitted whenever $c_i = 0$. For instance, the marking of the net in Fig. 3.3.a can be written as $\langle 3p_1 + 5p_2 + 2p_3 \rangle$. If places are ordered in some way, a marking can also be represented as a vector, in the given example it becomes $[3, 5, 2, 0]^T$.

A marking is said to be *valid* w.r.t. a certain net if no place contains more tokens than its capacity. A transition is said to be *enabled* if it satisfies its local triggering conditions: each place connected to the transition with an incoming arc shall contain a number of tokens greater than or equal to the arc instantaneous weight, and each place connected to the transition with an outgoing arc shall have enough space to accommodate the tokens prescribed by the instantaneous weight of the arc, i.e. the instantaneous weight plus the tokens already present has to be less than or equal to the place capacity. The execution of an enabled transition is described by the so called *firing rule*: a transition *fires* as a single atomic step that removes the tokens of each place connected with an incoming arc, and it adds new tokens in each place connected with an outgoing arc. The number of removed or

added tokens is determined by the instantaneous weight $w \cdot \hat{p}$ of the corresponding arcs, namely w times the tokens currently stored in p in the enabling state.

The firing rule is exemplified in Fig. 3.3 by means of an intermediate phase showing the instantaneous weight of each arc: Fig. 3.3.a represents the net in the initial state, Fig. 3.3.b shows how instantaneous weights are computed, then the triggering conditions are evaluated; in particular, p_1 contains more than 1 tokens, p_2 contains more than 4 tokens and the tokens added to p_3 and p_4 do not exceed their capacities, hence t_1 is enabled and can fire. The resulting state of the firing is depicted in Fig. 3.3.c.

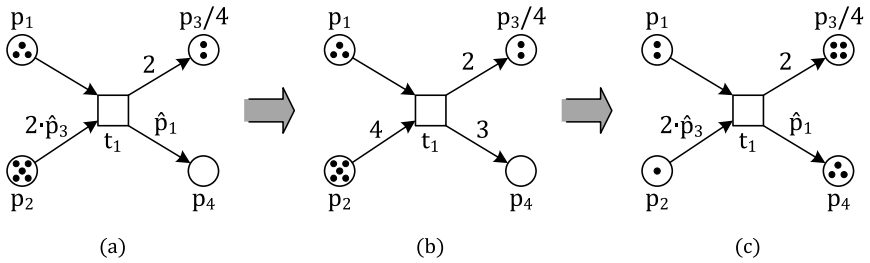


Fig. 3.3. Example of the PTNs firing rule. (a) The original net configuration. (b) An intermediate phase after the evaluation of all instantaneous weights. (c) The new net configuration after the evaluation of the triggering conditions and the firing of t_1.

The label ℓ associated to each transition can be interpreted as a *visible action*, offered to the environment, i.e. the system waits indefinitely in the current state until an external observer chooses one of the enabled actions. The only exception are those transitions labeled with the special silent action τ that are executed internally without the help of the environment.

The overall behavior of a net is given by the full range of interaction patterns offered to an external observer that in turn depends on the visible actions available on each reachable state. A good metaphor is given by a machine with buttons [59]: an action can be seen as a labeled button, when the underlying transition is enabled the button lights up and can be pressed by the user, otherwise the button is turned off and cannot be pressed.

In any valid state, zero or more atomic transitions may be enabled at the same time, but only one of them can fire, potentially altering the current token distribution. If no transition is enabled in a certain state, such state is said to be *terminal* and no further interactions are possible. If only one transition is enabled, the interpretation is trivial and depends on the transition label: in case of a silent action the system fires it and goes on, otherwise it waits until the only offered action is performed. If more than one transition is enabled, the interpretation becomes more complex, because visible actions may be duplicated and they may be also mixed with silent ones. Several yet reasonable scheduling strategies can be defined to face such case; for instance, the less frequently performed transition can be chosen by tracking the number of firings of each transition. Another interesting aspect is proving that the external visible behavior of a system does not depend on the scheduling of silent transitions, and so on.

Anyway, this chapter is only intended as an introduction to Petri nets: therefore, for sake of simplicity the focus is on a particular kind of nets with an easy to understand interpretation, which is identified here by the notion of plain net. A net is said to be *simple* if there is no state, reachable from the initial one, that enables two transitions with the same action, including silent ones. A transition is atomic and takes no time to fire unless it waits for an action from the environment: hence, it can be assumed that the interpreter can run as many silent actions as possible until a stable waiting state is reached. In general, it is not guaranteed that such waiting state exists, i.e. the net can enter into an infinite internal loop. In a simple net the interpreter can fire at most one silent transition at time, hence no complex scheduling is required to choose the next internal step.

Example 3.1. A first example of process model specified in PTNs is the readers-writers system in Fig. 3.4 originally proposed in [14] and reported here with some minor changes. The net is clearly simple since any transition has a distinct action. Such net describes a process involving $x \in \mathbb{N}$ programs that can access to a shared portion of the memory.

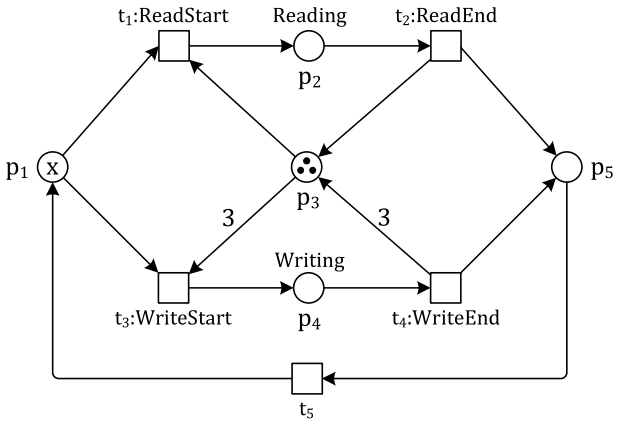


Fig. 3.4. A readers-writers system originally proposed in [14] and modeled here as a PTNs net with some minor changes, in particular a fixed number of system resources w.r.t. the running programs x and a silent transition t_5.

Up to 3 running programs can simultaneously read the shared memory, but only one program at time can write it and only when no other ones are in the reading state. When t_1 fires, 1 out of x programs enters in the reading state removing one token from p_3. Since t_3 requires 3 tokens from p_3 to fire, no program can write the shared memory while someone is reading it. Conversely, when t_3 fires all tokens in p_3 are removed until the program does not exit from the writing state, hence no other program can enter in the reading state by firing t_1. The number of programs in p_1 is restored through the silent transition t_5 that fires promptly as soon as a token is placed in p_5.

Notice that the net in Fig. 3.4 models only quantitative aspects of the system, because there is no way to distinguish individual tokens, e.g. it is not relevant which program exits first from the reading state w.r.t. the entering order. □

The model in Fig. 3.4 discussed in Ex. 3.1 is made only with arcs of constant weights, the next example introduces a simple net that takes advantage of instantaneous weights.

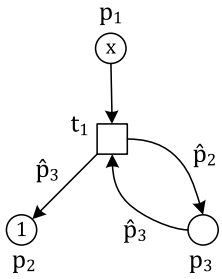
Example 3.2. Given a number $x \in \mathbb{N}$, the recursive function $fib : \mathbb{N} \to \mathbb{N}$ defined in Eq. 3.1 returns the x-element of the Fibonacci sequence starting from 1. If it is desired, the 0 element can be included in the sequence with slightly modification of the recursive definition, while the algorithm and the net in Fig. 3.5 do not change.

$$\forall x \in \mathbb{N} \,.\, fib(x) = \begin{cases} 1 & \text{if } x \leq 2 \\ fib(x-1) + fib(x-2) & \text{otherwise} \end{cases} \tag{3.1}$$

The function Eq. 3.1 is implemented in Fig. 3.5.a as an iterative algorithm using three temporary variables. The same function is implemented with a PTNs model where the argument is placed in p_1 and the result is the number of tokens in p_3 when p_1 becomes empty.

```
y ← FIB(x)
1  y ← 0
2  z ← 1
3  while x > 0 do
4        w ← y
5        y ← z
6        z ← z + w
7        x ← x − 1
8  end while
9  return y
```

(a)



(b)

	p_1	p_2	p_3
	6	1	0
t_1	5	1	1
t_1	4	2	1
t_1	3	3	2
t_1	2	5	3
t_1	1	8	5
t_1	0	13	8

(c)

Fig. 3.5. (a) An iterative algorithm to compute Fibonacci numbers. (b) A net that computes Fibonacci numbers, the result is the number of tokens that can be read from p_3 when the net halts because $\hat{p_1} = 0$. (c) A complete run of the net for $x = 6$: in each state, only t_1 can be fired, hence the execution is easily representable in tabular form.

From the net in Fig. 3.5 it is easy to obtain a net able to compute the Fibonacci sequence by reversing the arc (p_1, t_1): with the reversed arc, t_1 can be fire indefinitely from the initial state $[0, 1, 0]^T$. After $i \in \mathbb{N}$ firings of t_1, the marking will be on the form $[i, fib(i+1), fib(i)]^T$. □

3.2.3 Formal Semantics

In this section the PTNs bare language semantics is formally stated in terms of mathematical functions. The first part of the section defines what is a net, then it introduces the notion of state or marking and finally, how a net is interpreted.

For obtaining more readable function signatures, three generic sets $\mathcal{P}, \mathcal{T}, \mathcal{A} \subseteq \mathcal{U}$ are introduced containing the possible identifiers for places, transitions and actions, respectively. These sets are not further specified, it is only assumed that $\theta \notin \mathcal{P}$ and $\tau \notin \mathcal{A}$: in other words the ideal place θ cannot be used as a place identifier, and the silent action τ cannot be used as an action identifier.

Definition 3.3 (Place Transition Nets). A *place transition net* is a tuple $\langle P, T, F \rangle$ that belongs to the set $\mathcal{N}$ defined as follows:

$$\begin{aligned} &\forall a \in \mathcal{U} \,.\, a \in \mathcal{N} \triangleq \\ &\quad a = \langle P, T, F \rangle \wedge \\ &\quad P : \mathcal{P} \rightharpoonup \mathbb{N} \cup \{\omega\} \;\wedge\; T : \mathcal{T} \rightharpoonup \mathcal{A} \cup \{\tau\} \;\wedge \\ &\quad V = dom(P) \cup dom(T) \;\wedge\; |V| = |dom(P)| + |dom(T)| < \omega \wedge \\ &\quad F : V \times V \rightharpoonup \mathbb{N} \times (dom(P) \cup \{\theta\}) \end{aligned} \tag{3.2}$$

where: $\langle V, dom(F) \rangle \in \mathcal{G}$ is a finite directed graph such that $\theta \notin V$, since it is a reserved symbol that cannot be used as vertex identifier. $P : \mathcal{P} \rightharpoonup \mathbb{N} \cup \{\omega\}$ is a partial function that for every place in the net returns its capacity. The symbol ω denotes the absence of a finite capacity. $T : \mathcal{T} \rightharpoonup \mathcal{A} \cup \{\tau\}$ is a function that for every transition in the net returns its action, eventually the silent one. Finally, the function $F : V \times V \rightharpoonup \mathbb{N} \times (dom(P) \cup \{\theta\})$ stores for each arc its instantaneous weight. The symbol θ denotes an ideal place that always contains one and only one token independently from the current state.

Table 3.1. Basic operations on PTNs nets.

Symbol	Function	Description	Ref
	$places : \mathcal{N} \to \wp(\mathcal{P} \times \mathbb{N} \cup \{\omega\})$	Places with their capacities.	*inline*
	$transitions : \mathcal{N} \to \wp(\mathcal{T} \times \mathcal{A} \cup \{\tau\})$	Transitions with their action.	*inline*
	$arcs : \mathcal{N} \to \wp(\mathcal{U} \times \mathcal{U} \times \mathbb{N} \times \mathcal{P} \cup \{\theta\})$	Net arcs with their weight.	*inline*
	$pls : \mathcal{N} \to \wp(\mathcal{P})$	Vertices that are places.	*inline*
	$trs : \mathcal{N} \to \wp(\mathcal{T})$	Vertices that are transitions.	*inline*
	$vertices : \mathcal{N} \to \wp(\mathcal{U})$	Vertices of the underlying graph.	*inline*
	$edges : \mathcal{N} \to \wp(\mathcal{U} \times \mathcal{U})$	Edges of the underlying graph.	*inline*
ζ	$capacity : \mathcal{N} \times \mathcal{P} \rightharpoonup \mathbb{N} \cup \{\omega\}$	Place capacity.	*inline*
λ	$action : \mathcal{N} \times \mathcal{T} \rightharpoonup \mathcal{A}$	Action associated to a transition.	*inline*
Λ	$actions : \mathcal{N} \to \wp(\mathcal{A})$	Set of declared actions.	*inline*
	$weight : \mathcal{N} \times \mathcal{U} \times \mathcal{U} \rightharpoonup \mathbb{N} \cup \{0\}$	Weight coefficient of an arc.	Eq. 3.3

The basic access functions are summarized in Tab. 3.1. In particular, a PTNs net has three fields $fields(\mathcal{N}) = \{places, transitions, arcs\}$ with signature

$places : \mathcal{N} \to \wp(\mathcal{P} \times \mathbb{N} \cup \{\omega\})$, $transitions : \mathcal{N} \to \wp(\mathcal{T} \times \mathcal{A} \cup \{\tau\})$, and $arcs : \mathcal{N} \to \wp(\mathcal{U} \times \mathcal{U} \times \mathbb{N} \times \mathcal{P} \cup \{\theta\})$. For every $a \in \mathcal{N}$ such that $a = \langle P, T, F \rangle$, $places(a) = P$, $transitions(a) = T$, and $arcs(a) = F$. The function $pls : \mathcal{N} \to \wp(\mathcal{P})$ returns the set of place identifiers contained in the net, which can be formalized as $pls(a) = dom(places(a))$. Similarly, the function $trs : \mathcal{N} \to \wp(\mathcal{U})$ stores the set of transition identifiers used in the net and corresponds to $trs(a) = dom(transitions(a))$. The functions $vertices : \mathcal{N} \to \wp(\mathcal{U})$ and $edges : \mathcal{N} \to \wp(\mathcal{U} \times \mathcal{U})$ return the set of vertices and edges of the underlying graph, respectively. They are formally defined as follows: $vertices(a) = pls(a) \cup trs(a)$, while $edges(a) = dom(arcs(a))$.

The $capacity : \mathcal{N} \times \mathcal{P} \nrightarrow \mathbb{N} \cup \{\omega\}$ operator returns the capacity associated to each place in the net. It is defined only on existing places as follows: $\forall a \in \mathcal{N} \,.\, \forall p \in pls(a) \,.\, capacity(a, p) = k$ if $(p, k) \in places(a)$, and is undefined otherwise. In similar way, $action : \mathcal{N} \times \mathcal{T} \nrightarrow \mathcal{A} \cup \{\tau\}$ is defined only on existing transitions as $\forall a \in \mathcal{N} \,.\, \forall t \in trs(a) \,.\, action(a, t) = \ell$ if $(t, \ell) \in transitions(a)$, and is undefined otherwise. The symbols ζ and λ are used as shorthand for *capacity* and *action*, respectively. Function $actions : \mathcal{N} \to \wp(\mathcal{A})$ returns the set of actions declared in the net: they are obtained as $\forall a \in \mathcal{N} \,.\, actions(a) = range(transitions(a)) \setminus \{\tau\}$, and briefly denoted using the auxiliary symbol Λ.

The partial function $weight : \mathcal{N} \times \mathcal{U} \times \mathcal{U} \nrightarrow \mathbb{N} \cup \{0\}$ is defined in Eq. 3.3: it returns the weight coefficient of the arc if such arc exists, and is undefined otherwise. Therefore, it can be used to check the presence of an arc.

$$
\begin{aligned}
&weight : \mathcal{N} \times \mathcal{U} \times \mathcal{U} \nrightarrow \mathbb{N} \cup \{0\} \;\; is \\
&\quad \forall a \in \mathcal{N} \,.\, \forall \langle x, y \rangle \in \mathcal{U} \times \mathcal{U} \,. \\
&\qquad weight(a, x, y) = \begin{cases} w & \text{if } \exists w \in \mathbb{N} \,.\, \exists p \in pls(a) \cup \{\theta\} \,.\, (x, y, w, p) \in arcs(a) \\ 0 & \text{if } x, y \in vertices(a) \\ \uparrow & \text{otherwise} \end{cases}
\end{aligned}
\tag{3.3}
$$

The functions $pre\text{-}set : \mathcal{N} \times \mathcal{U} \to \wp(\mathcal{U})$ and $post\text{-}set : \mathcal{N} \times \mathcal{U} \to \wp(\mathcal{U})$ in Tab. 3.2 become useful to capture the locality of a transition in order to check its enabling conditions.

Table 3.2. More operations on PTNs nets.

Symbol	Function	Description	Ref
	$pre\text{-}set : \mathcal{N} \times \mathcal{U} \nrightarrow \wp(\mathcal{U})$	Elements connected to the specified vertex with an incoming arc.	Eq. 3.4
	$post\text{-}set : \mathcal{N} \times \mathcal{U} \nrightarrow \wp(\mathcal{U})$	Elements connected to the specified vertex with an outgoing arc.	Eq. 3.5
	$components : \mathcal{N} \times \mathcal{U} \nrightarrow \wp(\mathcal{U})$	It return the vertices reachable from the specified vertex.	Eq. 3.6
	$union : \mathcal{N} \times \mathcal{N} \to \mathcal{N}$	Union between two nets.	Eq. 3.7

These functions are defined over all vertices: hence, they can also be applied to a place for obtaining the surrounding transitions. In particular, the pre-set of a transition t is defined as the set of all places that have an incoming arc in t. In the

same way, the pre-set of a place p is the set of all transitions having an incoming arc in p. The post-set is defined in a similar way for both places and transitions.

$$\begin{aligned} &\mathit{pre\text{-}set} : \mathcal{N} \times \mathcal{U} \rightharpoonup \wp(\mathcal{U}) \ \mathit{is} \\ &\quad \forall a \in \mathcal{N} \,.\, \forall u \in \mathcal{U} \,. \\ &\qquad \mathit{pre\text{-}set}(a,u) = \begin{cases} \{v \in \mathit{vertices}(a) \mid (v,u) \in \mathit{edges}(a)\} & \text{if } u \in \mathit{vertices}(a) \\ \uparrow & \text{otherwise} \end{cases} \end{aligned} \tag{3.4}$$

$$\begin{aligned} &\mathit{post\text{-}set} : \mathcal{N} \times \mathcal{U} \rightharpoonup \wp(\mathcal{U}) \ \mathit{is} \\ &\quad \forall a \in \mathcal{N} \,.\, \forall u \in \mathcal{U} \,. \\ &\qquad \mathit{post\text{-}set}(a,u) = \begin{cases} \{v \in \mathit{vertices}(a) \mid (u,v) \in \mathit{edges}(a)\} & \text{if } u \in \mathit{vertices}(a) \\ \uparrow & \text{otherwise} \end{cases} \end{aligned} \tag{3.5}$$

The partial function $\mathit{components} : \mathcal{N} \times \mathcal{U} \rightharpoonup \wp(\mathcal{U})$ returns the set of vertices that are reachable from the specified vertex using the existing net arcs. Its formal definition is given in Eq. 3.6.

$$\begin{aligned} &\mathit{components} : \mathcal{N} \times \mathcal{U} \rightharpoonup \wp(\mathcal{U}) \ \mathit{is} \\ &\quad \forall a \in \mathcal{N} \,.\, \forall v \in \mathit{vertices}(a) \,.\, \forall u \in \mathcal{U} \,. \\ &\qquad u \in \mathit{components}(a,v) \stackrel{\triangle}{\Longleftrightarrow} \\ &\qquad\quad u = v \vee \exists z \in \mathit{components}(a,v) \,.\, (z,u) \in \mathit{edges}(a) \end{aligned} \tag{3.6}$$

Function $\mathit{union} : \mathcal{N} \times \mathcal{N} \to \mathcal{N}$ performs the union between two nets, as formally defined in Eq. 3.7.

$$\begin{aligned} &\mathit{union} : \mathcal{N} \times \mathcal{N} \to \mathcal{N} \ \mathit{is} \\ &\quad \forall a,b,c \in \mathcal{N} \,.\, c = \mathit{union}(a,b) \stackrel{\triangle}{\Longleftrightarrow} \\ &\qquad \mathit{vertices}(a) \cap \mathit{vertices}(b) = \varnothing \\ &\qquad \forall f \in \mathit{fields}(\mathcal{N}) \,.\, f(c) = f(a) \cup f(b) \end{aligned} \tag{3.7}$$

Markings

The notion of marking captures the state of a net. The marking definitions in Petri nets literature are often decoupled from a particular net instance, i.e. the state of a net can be also a valid state of another one. This can be obtained by implicitly assuming that any place not considered in a marking is empty. Here a more pragmatic approach is adopted: a marking of a net is defined as a function with a finite domain that has at least to declare the content of all net places.

Definition 3.4 (Marking). Assuming that place identifiers are taken from the set $\mathcal{P}$, a marking is a partial function $q : \mathcal{P} \rightharpoonup \mathbb{N} \cup \{0\}$ of finite domain that for each declared place returns the number of tokens in it and always returns 1 for the ideal place θ. The set of all markings is denoted by $\mathcal{Q}$ defined as:

$$\mathcal{Q} = \{\langle f \rangle \mid f : \mathcal{P} \rightharpoonup \mathbb{N} \cup \{0\} \ \wedge \ |\mathit{dom}(f)| < \omega \wedge f(\theta) = 1\} \tag{3.8}$$

Many useful operations can be defined on markings, the basic ones are summarized in Tab. 3.3. In particular, a marking has only one field for accessing its internal representation $\mathit{fields}(\mathcal{Q}) = \{\mathit{body}\}$. Such function is defined as follows: for every marking $q \in \mathcal{Q}$ such that $q = \langle f \rangle$, $\mathit{body}(q) = \langle f \rangle$.

Table 3.3. Common operations on markings.

Notation	Function	Description	Ref
	$\mathit{body} : \mathcal{Q} \to \wp(\mathcal{P} \times \mathbb{N} \cup \{0\})$	Internal representation.	*inline*
	$\mathit{count} : \mathcal{Q} \times \mathcal{P} \rightharpoonup \mathbb{N} \cup \{0\}$	Tokens in the given place.	Eq. 3.9
	$\mathit{marked} : \mathcal{Q} \to \wp(\mathcal{U})$	Marked places.	*inline*
$\lvert \cdot \rvert$	$\mathit{size} : \mathcal{Q} \to \mathbb{N} \cup \{0\}$	The number of marked places.	*inline*
$\cdot \in \cdot$	$\mathit{in} : \mathcal{U} \times \mathcal{Q} \to \mathbb{B}$	Check if a place is marked.	3.10
$\cdot \geq \cdot$	$\mathit{geq} : \mathcal{Q} \times \mathcal{Q} \to \mathbb{B}$	Greater than or equal to relation.	3.11
$\cdot > \cdot$	$\mathit{ge} : \mathcal{Q} \times \mathcal{Q} \to \mathbb{B}$	Greater than relation.	3.12
$\cdot \leq \cdot$	$\mathit{leq} : \mathcal{Q} \times \mathcal{Q} \to \mathbb{B}$	Less than or equal to relation.	3.13
$\cdot < \cdot$	$\mathit{le} : \mathcal{Q} \times \mathcal{Q} \to \mathbb{B}$	Less than relation.	3.14

Since a marking is represented as a function with a finite domain, it can be defined by enumerating its values: given a net $a \in \mathcal{N}$ with $\mathit{pls}(a) \triangleq \{p_i\}_{i=1}^{m}$, a marking $q \in \mathcal{Q}$ can be declared as $q \triangleq \{p_i \mapsto v_i\}_i^m$ letting by convention the angle brackets, the element $\langle \theta, 1 \rangle$ and any element $\langle p_i, 0 \rangle$ implicit. This is an alternative mathematical notation to the usual polynomial-like notation which expresses q as $\langle \sum_{i=0}^{m} v_i p_i \rangle$ for every $v_i \neq 0$. Whenever the current state $q \in \mathcal{Q}$ is clear from the context, e.g. it is graphically represented as tokens in the net, the content of a place p_i is denoted as $\hat{p_i}$. The intuitive notation $q(x)$ is adopted to obtain the number of tokens inside the place x that is captured by the simple function $\mathit{count} : \mathcal{Q} \times \mathcal{P} \rightharpoonup \mathbb{N} \cup \{0\}$ defined as:

$$\begin{aligned} &\mathit{count} : \mathcal{Q} \times \mathcal{P} \rightharpoonup \mathbb{N} \cup \{0\} \ \mathit{is} \\ &\quad \forall q \in \mathcal{Q} \,.\, \forall p \in \mathit{marked}(q) \,.\, \forall n \in \mathbb{N} \cup \{0\} \,. \\ &\qquad n = \mathit{count}(q, p) \iff (p, n) \in \mathit{body}(q) \end{aligned} \tag{3.9}$$

The domain of a marking excluding θ is captured by the function $\mathit{marked} : \mathcal{Q} \to \wp(\mathcal{U})$ such that for any $q \in \mathcal{Q}$, $\mathit{marked}(q) = \mathit{dom}(\mathit{body}(q)) \setminus \{\theta\}$, while the size of a marking $\mathit{size} : \mathcal{Q} \to \mathbb{N} \cup \{0\}$ is defined as the number of its explicit places, i.e. $|q| = \mathit{size}(q) = |\mathit{marked}(q)|$. The inclusion relation $\cdot \in \cdot$ can be consistently overloaded to check if a marking declares the number of tokens of a certain place: such function is defined in Eq. 3.10.

$$\begin{aligned} &\mathit{in} : \mathcal{U} \times \mathcal{Q} \to \mathbb{B} \ \mathit{is} \\ &\quad \forall u \in \mathcal{U} \,.\, \forall q \in \mathcal{Q} \,.\, \mathit{in}(u, q) = \begin{cases} \mathsf{T} & \text{if } u \in \mathit{marked}(q) \\ \mathsf{F} & \text{otherwise} \end{cases} \end{aligned} \tag{3.10}$$

The remaining relations are defined in Eq. 3.11 through Eq. 3.14. Each one overloads an existing mathematical symbol in a fairly intuitive way. The negation

of such symbols are also used for reversing the operation result; for instance, $\forall q, r \in \mathcal{Q} \,.\, q \not\geq r \Leftrightarrow \neg(q \geq r)$.

$$geq : \mathcal{Q} \times \mathcal{Q} \to \mathbb{B} \;\; is \tag{3.11}$$

$$\forall q, r \in \mathcal{Q} \,.\, geq(q, r) = \begin{cases} \mathsf{T} & \text{if } \forall u \in marked(q) \cap marked(r) \,.\, q(u) \geq r(u) \;\wedge \\ & \quad \forall u \in marked(r) \setminus marked(q) \,.\, r(u) = 0 \\ \mathsf{F} & \text{otherwise} \end{cases}$$

$$ge : \mathcal{Q} \times \mathcal{Q} \to \mathbb{B} \;\; is \tag{3.12}$$

$$\forall q, r \in \mathcal{Q} \,.\, ge(q, r) = \begin{cases} \mathsf{T} & \text{if } q \geq r \;\wedge \\ & \quad (\, \exists x \in marked(q) \cap marked(r) \,.\, q(x) > r(x) \;\vee \\ & \quad \exists y \in marked(q) \setminus marked(r) \,.\, q(y) \neq 0 \,) \\ \mathsf{F} & \text{otherwise} \end{cases}$$

$$leq : \mathcal{Q} \times \mathcal{Q} \to \mathbb{B} \;\; is \tag{3.13}$$

$$\forall q, r \in \mathcal{Q} \,.\, leq(q, r) = \begin{cases} \mathsf{T} & \text{if } \forall u \in marked(q) \cap marked(r) \,.\, q(u) \leq r(u) \;\wedge \\ & \quad \forall u \in marked(q) \setminus marked(r) \,.\, q(u) = 0 \\ \mathsf{F} & \text{otherwise} \end{cases}$$

$$le : \mathcal{Q} \times \mathcal{Q} \to \mathbb{B} \;\; is \tag{3.14}$$

$$\forall q, r \in \mathcal{Q} \,.\, le(q, r) = \begin{cases} \mathsf{T} & \text{if } q \leq r \;\wedge \\ & \quad (\, \exists x \in marked(q) \cap marked(r) \,.\, q(x) < r(x) \;\vee \\ & \quad \exists y \in marked(r) \setminus marked(q) \,.\, r(y) \neq 0 \,) \\ \mathsf{F} & \text{otherwise} \end{cases}$$

These definitions are not redundant: it is easy to prove that $\forall q, r \in \mathcal{Q} \,.\, q \leq r \Rightarrow q \not> r$, but the contrary $\forall q, r \in \mathcal{Q} \,.\, q \not\leq r \Rightarrow q > r$ does not hold. For instance, the two markings $q \triangleq \{p_1 \mapsto 0, p_2 \mapsto 1\}$ and $r \triangleq \{p_1 \mapsto 1, p_2 \mapsto 0\}$ cannot be compared: $q \not\geq r$ because $q(p_1) < r(p_1)$, as a consequence $q \not> r$; conversely $q \not\leq r$ because $q(p_2) \not> r(p_2)$ and this implies also $q \not< r$.

Table 3.4. Common operations on markings.

Notation	Function	Description	Ref
	$zero : \mathcal{N} \to \mathcal{Q}$	Empty marking for the given net.	*inline*
$\cdot + \cdot$	$add : \mathcal{Q} \times \mathcal{Q} \rightharpoonup \mathcal{Q}$	Add two markings.	3.15
$\cdot \ominus \cdot$	$subtract : \mathcal{Q} \times \mathcal{Q} \rightharpoonup \mathcal{Q}$	Subtract two markings.	3.16
$\lVert \cdot \rVert$	$norm : \mathcal{Q} \to \mathbb{N} \cup \{0\}$	Total number of tokens in the given marking.	3.17

Some common operations on markings that preserve their properties are summarized in Tab. 3.4. For every net $a \in \mathcal{N}$, the function $zero : \mathcal{N} \to \mathcal{Q}$ returns a marking $q \in \mathcal{Q}$ such that for any $u \in \mathcal{U}$, $q(\theta) = 1$, $q(u) = 0$ for all $u \in pls(a)$ and $q(u) \uparrow$ otherwise. Addition, safe substraction, and norm are defined in Eq. 3.15, Eq. 3.16, and Eq. 3.17, respectively.

$$add : \mathcal{Q} \times \mathcal{Q} \rightharpoonup \mathcal{Q} \ is \tag{3.15}$$
$$\forall q, r \in \mathcal{Q} \,.\, add(q,r) = s \in \mathcal{Q} \,.$$
$$\forall u \in \mathcal{U} \,.\, s(u) = \begin{cases} 1 & \text{if } u = \theta \\ q(u) + r(u) & \text{if } u \in q \wedge u \in r \wedge u \neq \theta \\ q(u) & \text{if } u \in q \wedge u \notin r \\ r(u) & \text{if } u \notin q \wedge u \in r \\ \uparrow & \text{otherwise} \end{cases}$$

$$subtract : \mathcal{Q} \times \mathcal{Q} \rightharpoonup \mathcal{Q} \ is \tag{3.16}$$
$$\forall q, r \in \mathcal{Q} \,.\, subtract(q,r) = s \in \mathcal{Q} \,.$$
$$\forall u \in \mathcal{U} \,.\, s(u) = \begin{cases} 1 & u = \theta \\ max\,\{q(u) - r(u), 0\} & \text{if } u \in q \wedge u \in r \wedge u \neq \theta \\ q(u) & \text{if } u \in q \wedge u \notin r \\ 0 & \text{if } u \notin q \wedge u \in r \\ \uparrow & \text{otherwise} \end{cases}$$

$$norm : \mathcal{Q} \to \mathbb{N} \cup \{0\} \ is \tag{3.17}$$
$$\forall q \in \mathcal{Q} \,.\, norm(q) = \sum_{x \in marked(q)} q(x)$$

In particular, $norm : \mathcal{Q} \to \mathbb{N} \cup \{0\}$ returns the total number of tokens in its marked places, ideal place not included, e.g. for all net $a \in \mathcal{N}$ $\|zero(a)\| = 0$.

Definition 3.5 (Valid Marking). A marking is said to be *valid* for a given net $a \in \mathcal{N}$ if and only if it is defined for each place of the net and its value does not exceed the place capacities. The set of all valid markings of a net is denoted by the function $markings : \mathcal{N} \to \wp(\mathcal{Q})$ defined as follows

$$\forall a \in \mathcal{N} \,.\, markings(a) \triangleq \tag{3.18}$$
$$\{ q \in \mathcal{Q} \mid \forall p \in pls(a) \,.\, p \in marked(q) \wedge \ q(p) \leq capacity(a,p) \}$$

Definition 3.6 (Marked Net). A marked net is a pair $\langle a, q \rangle$ such that the former element is a net $a \in \mathcal{N}$ and the latter one is a valid marking $q \in markings(a)$. The symbol $\mathcal{N}^\bullet$ is used to denote the set of all marked nets which is defined as follows:

$$\mathcal{N}^\bullet \triangleq \{ \langle a, q \rangle \mid a \in \mathcal{N} \wedge q \in markings(a) \} \tag{3.19}$$

Notice that for all $a \in \mathcal{N}$, the pair $\langle a, zero(a) \rangle \in \mathcal{N}^\bullet$, hence $\mathcal{N}$ can be improperly considered a subset of $\mathcal{N}^\bullet$; similarly, given a net $a \in \mathcal{N}^\bullet$ such that $a = \langle b, q \rangle$, the first element $b \in \mathcal{N}$ can be seen as the *structure* of a. The functions $net : \mathcal{N}^\bullet \to \mathcal{N}$ and $initial\text{-}marking : \mathcal{N}^\bullet \to \mathcal{Q}$ capture such difference: for any $\langle b, q \rangle \in \mathcal{N}^\bullet$, $net(b,q) = b$ and $initial\text{-}marking(b,q) = q$. The $net : \mathcal{N}^\bullet \to \mathcal{N}$ operator is left implicit whenever a function accepting a net is used on a marked net, e.g. we can write $pls(a)$ instead of $pls(net(a))$ when a is marked.

Formal Interpretation

This section explains how a net is interpreted. A run of a net is essentially a sequence of firings driven by external actions. At each step, the available actions are a function of the current internal state which is altered by the performed transitions.

Definition 3.7 (Instantaneous Weight). The *instantaneous weight* of an arc depends on the current state and it is defined as follows:

$$\begin{aligned} &\mathit{weight} : \mathcal{N} \times \mathcal{U} \times \mathcal{U} \times \mathcal{Q} \rightharpoonup \mathbb{N} \cup \{0\} \ \mathit{is} \\ &\quad \forall a \in \mathcal{N} \,.\, \forall (u,v) \in \mathcal{U} \times \mathcal{U} \,.\, \forall q \in \mathit{markings}(a) \,. \\ &\qquad \mathit{weight}(a,u,v,q) = \begin{cases} w \cdot q(p) & \text{if } \exists w \in \mathbb{N} \,.\, \exists p \in \mathit{pls}(a) \cup \{\theta\} \ .\, (u,v,w,p) \in \mathit{arcs}(a) \\ 0 & \text{if } u,v \in \mathit{vertices}(a) \\ \uparrow & \text{otherwise} \end{cases} \end{aligned} \tag{3.20}$$

The function is defined only for valid markings. For every net $a \in \mathcal{N}$, if the argument $(u,v) \notin \mathit{edges}(a)$, the function evaluates to 0. It worths noting that when the arc is constant, its instantaneous weight is equivalent to its weight coefficient:

$$\begin{aligned} &\forall a \in \mathcal{N} \,.\, \forall (u,v) \in \mathit{edges}(a) \,.\, \forall q \in \mathit{markings}(a) \,. \\ &\quad \mathit{is\text{-}const}(a,u,v) \Rightarrow \mathit{weight}(a,u,v,q) = \mathit{weight}(a,u,v) \end{aligned} \tag{3.21}$$

Definition 3.8 (Enabled Set). A transition $t \in \mathit{trs}(a)$ of a net $a \in \mathcal{N}$ is said to be *enabled* in $q \in \mathit{markings}(a)$ if each place $x \in \mathit{pre\text{-}set}(t)$ contains a number of tokens $q(x)$ greater than or equals to the instantaneous weight $\mathit{weight}(a,x,t,q)$, and each place $y \in \mathit{post\text{-}set}(t)$ has enough space to store new tokens. The space is enough if the existing tokens $q(y)$ plus the new ones $\mathit{weight}(a,t,y,q)$ are less than or equal to the capacity $\mathit{capacity}(a,y)$ minus the removed tokens, due to $\mathit{weight}(a,y,t,q)$ if the corresponding arc exists. The *enabled set* of a net $a \in \mathcal{N}$ is the set of all transitions $t \in \mathit{trs}(a)$ which are enabled in a given marking $q \in \mathit{markings}(a)$. The enabled-set is captured by the function $\mathit{enabled\text{-}set} : \mathcal{N} \times \mathcal{U} \to \wp(\mathcal{U})$ defined as:

$$\begin{aligned} &\forall a \in \mathcal{N} \,.\, \forall q \in \mathit{markings}(a) \,.\, \forall t \in \mathit{trs}(a) \,. \\ &\quad t \in \mathit{enabled\text{-}set}(a,q) \overset{\triangle}{\Longleftrightarrow} \\ &\quad \forall x \in \mathit{pre\text{-}set}(a,t) \ .\, q(x) \geq \mathit{weight}(a,x,t,q) \wedge \\ &\quad \forall y \in \mathit{post\text{-}set}(a,t) \,.\, q(y) \leq \zeta(a,y) - \mathit{weight}(a,t,y,q) + \mathit{weight}(a,y,t,q) \end{aligned} \tag{3.22}$$

An enabled transition can *fire* potentially changing the current state of the net. The concept of firing a transition to compute the next state is captured by the following definition.

Definition 3.9 (Next State). Given a net $a \in \mathcal{N}$, a transition $t \in \mathit{trs}(a)$ which is enabled in a state $q \in \mathit{markings}(a)$, can fire moving the system from q to a next state $r \in \mathit{markings}(a)$, such that the necessary amount of tokens in $\mathit{pre\text{-}set}(a,t)$ have been consumed and some new tokens have been produced in $\mathit{post\text{-}set}(a,t)$. Formally the firing of a transition is defined as follows:

$$
\begin{aligned}
&\mathit{fire} : \mathcal{N} \times \mathcal{Q} \times \mathcal{U} \rightharpoonup \mathcal{Q} \\
&\quad \forall a \in \mathcal{N} \,.\, \forall q \in \mathit{markings}(a) \,.\, \forall t \in \mathit{enabled\text{-}set}(a,q) \,.\, \forall r \in \mathcal{Q} \\
&\qquad \mathit{fire}(a,q,t) = r \mathrel{\triangleq\!\!\Longleftrightarrow} \forall x \in \mathit{pls}(a) \,. \\
&\qquad\qquad r(x) = \begin{cases} q(x) - \mathit{weight}(a,x,t,q) + & \text{if } x \in \mathit{pre\text{-}set}(a,t) \cup \mathit{post\text{-}set}(a,t) \\ \qquad + \mathit{weight}(a,t,x,q) & \\ q(x) & \text{if } x \notin \mathit{pre\text{-}set}(a,t) \cup \mathit{post\text{-}set}(a,t) \\ \uparrow & \text{otherwise} \end{cases}
\end{aligned}
\tag{3.23}
$$

Notice that the function $\mathit{fire} : \mathcal{N} \times \mathcal{Q} \times \mathcal{U} \rightharpoonup \mathcal{Q}$ is defined only for valid markings and only for enabled transitions.

Definition 3.10 (Reachability Set). Given a net $a \in \mathcal{N}$, a state $r \in \mathcal{Q}$ is said to be *reachable* from $q \in \mathcal{Q}$ if it belongs to the *reachability set* defined in Eq. 3.24:

$$
\begin{aligned}
&\forall a \in \mathcal{N} \,.\, \forall q \in \mathit{markings}(a) \,.\, \forall r \in \mathcal{Q} \,. \\
&\quad r \in \mathit{reachability\text{-}set}(a,q) \mathrel{\triangleq\!\!\Longleftrightarrow} \\
&\qquad r = q \;\vee\; \exists s \in \mathit{reachability\text{-}set}(a,q) \,.\, \exists t \in \mathit{enabled\text{-}set}(a,s) \,.\, r = \mathit{fire}(a,s,t)
\end{aligned}
\tag{3.24}
$$

The reachability set of a net $a \in \mathcal{N}$ starting from a marking $q \in \mathcal{Q}$ is shortly denoted as $\rho(a,q)$, while for a marked net $b \in \mathcal{N}^\bullet$ it becomes $\rho(b)$ that should be interpreted as $\rho(\mathit{net}(b), \mathit{initial\text{-}marking}(b))$.

In the following, some basic notions are introduced to describe the execution of a net in terms of fired transitions.

Definition 3.11 (Trace Set). A *trace* is a sequence of transitions. For convenience, the set of all possible finite traces $\mathit{trs}(a)^*$ of a net $a \in \mathcal{N}$ is denoted as $\mathcal{T}_R(a)$, while the generic set of all finite traces is denoted as $\mathcal{T}_R$:

$$
\mathcal{T}_R(a) = \mathit{trs}(a)^* \qquad \mathcal{T}_R = \bigcup_{a \in \mathcal{N}} \mathcal{T}_R(a) \subseteq \mathcal{T}^* \tag{3.25}
$$

Similarly, the set of all possible finite observable traces of a net $a \in \mathcal{N}$ is denoted as $\mathcal{T}_A(a)$, while the generic set of all finite observable traces is denoted as $\mathcal{T}_A$:

$$
\mathcal{T}_A(a) = \mathit{actions}(a)^* \qquad \mathcal{T}_A = \bigcup_{a \in \mathcal{N}} \mathcal{T}_A(a) \subseteq \mathcal{A}^* \tag{3.26}
$$

Definition 3.12 (Forward Firing Relation). Given a net $a \in \mathcal{N}$, the *forward firing relation* is the set of possible triples $\langle q, \sigma, r \rangle$ made of a marking $q \in \mathit{markings}(a)$, a trace $\sigma \in \mathcal{T}_R$ that can be executed by a starting from q, and the resulting marking $r \in \mathcal{Q}$ obtained by firing σ. It is formally defined as follows:

$$
\begin{aligned}
&\mathit{fr} : \mathcal{N} \to \wp(\mathcal{Q} \times \mathcal{T}_R \times \mathcal{Q}) \;\; \mathit{is} \\
&\quad \forall a \in \mathcal{N} \,.\, \forall q, r \in \mathcal{Q} \,.\, \forall \sigma \in \mathcal{T}_R \,. \\
&\qquad \langle q, \sigma, r \rangle \in \mathit{fr}(a) \mathrel{\triangleq\!\!\Longleftrightarrow} q \in \mathit{markings}(a) \;\wedge\; (r = q \vee \\
&\qquad\quad \exists s \in \rho(a,q) \,.\, \exists \eta \in \mathcal{T}_R \,.\, \exists t \in \mathcal{T} \,. \\
&\qquad\qquad t \in \mathit{enabled\text{-}set}(a,s) \,.\, \sigma = \eta \circ t \wedge (q, \eta, s) \in \mathit{fr}(a) \;\wedge r = \mathit{fire}(a,s,t))
\end{aligned}
\tag{3.27}
$$

A triple $\langle q, \sigma, r \rangle \in fr(a)$ can be denoted as $q \xrightarrow{\sigma} r$ when the underlying net $a \in \mathcal{N}$ can be inferred from the context, e.g. q is declared to belong to $markings(a)$. For sake of simplicity, when $\sigma = \langle t \rangle$, $q \xrightarrow{\langle t \rangle} r$ is denoted as $q \xrightarrow{t} r$.

Definition 3.13 (Net Traces). Given a net $a \in \mathcal{N}$ and an initial valid marking $q \in markings(a)$, a trace of a is a sequence of transitions $\sigma \in \mathcal{T}_R(a)$ that can be fired from q without interruption. The set of all finite transition traces of a net is captured by the function $traces : \mathcal{N} \times \mathcal{Q} \nrightarrow \wp(\mathcal{T}_R)$ defined as follows:

$$\begin{aligned} &traces : \mathcal{N} \times \mathcal{Q} \nrightarrow \wp(\mathcal{T}_R) \ \ is \\ &\quad \forall a \in \mathcal{N} \,.\, \forall q \in markings(a) \,.\, \forall \sigma \in \mathcal{T}_R \,. \\ &\qquad \sigma \in traces(a, q) \triangleq\!\!\Longleftrightarrow \exists r \in \rho(a, q) \,.\, \langle q, \sigma, r \rangle \in fr(a) \end{aligned} \tag{3.28}$$

Definition 3.14 (Action Sequence). Every transition $t \in trs(a)$ of a net $a \in \mathcal{N}$ is coupled with an action that can be obtained through the $action : \mathcal{N} \times \mathcal{T} \to \mathcal{A}$ function defined in Sec. 3.2.3 and represented by the symbol λ. Such function can be extended to traces as follows:

$$\begin{aligned} &\hat{\lambda} : \mathcal{N} \times \mathcal{T}_R \nrightarrow \mathcal{T}_A \ \ is \\ &\quad \forall a \in \mathcal{N} \,.\, \forall \sigma \in \mathcal{T}_R \,. \\ &\qquad \hat{\lambda}(a, \sigma) = \begin{cases} \varepsilon & \text{if } \sigma = \varepsilon \\ \lambda(a, t) \circ \hat{\lambda}(a, \eta) & \text{if } \exists t \in trs(a) \,.\, \exists \eta \in \mathcal{T}_R \,.\, \sigma = t \circ \eta \\ \uparrow & \text{otherwise} \end{cases} \end{aligned} \tag{3.29}$$

Definition 3.15 (Observable Trace). Given a net $a \in \mathcal{N}$ and a trace $\sigma \in \mathcal{T}_R$, its corresponding observable trace can be obtained using the function $obs : \mathcal{N} \times \mathcal{T}_R \to \mathcal{T}_A$, defined as the projection of $\hat{\lambda}(a, \sigma)$ w.r.t. observable actions:

$$\begin{aligned} &obs : \mathcal{N} \times \mathcal{T}_R \to \mathcal{T}_A \ \ is \\ &\quad \forall a \in \mathcal{N} \,.\, \forall \sigma \in \mathcal{T}_R \,.\, obs(a, \sigma) = \pi\left(actions(a), \hat{\lambda}(a, \sigma)\right) \end{aligned} \tag{3.30}$$

Definition 3.16 (Trace Firing). Given a net $a \in \mathcal{N}$, an initial valid marking $q \in markings(a)$, and a trace $\sigma \in \mathcal{T}_R$, the function $fire\text{-}trace : \mathcal{N} \times \mathcal{Q} \times \mathcal{T}_R \nrightarrow \mathcal{Q}$ returns the last state reached by firing the trace σ until it is possible:

$$\begin{aligned} &fire\text{-}trace : \mathcal{N} \times \mathcal{Q} \times \mathcal{T}_R \nrightarrow \mathcal{Q} \ \ is \\ &\quad \forall a \in \mathcal{N} \,.\, \forall q \in markings(a) \,.\, \forall \sigma \in \mathcal{T}_R \,. \\ &\qquad fire\text{-}trace(a, q, \sigma) = \begin{cases} fire(a, q, t) & \text{if } \sigma = \langle t \rangle \wedge t \in enabled\text{-}set(a, q) \\ fire\text{-}trace(a, r, \eta) & \text{if } \sigma = t \circ \eta \wedge \\ & \quad \wedge\, t \in enabled\text{-}set(a, q) \\ & \quad \wedge\, r = fire(a, q, t) \\ q & \text{otherwise} \end{cases} \end{aligned} \tag{3.31}$$

The *reachability graph* of a marked net $(a, q) \in \mathcal{N}^\bullet$ is an edge-labeled multi-graph not necessarily finite, such that each node represents a state s reachable from the initial one, i.e. $s \in \rho(a, q)$, two nodes s and r are connected with an edge if and only if there exists a transition t for which $\langle s, t, r \rangle \in fr(a)$, and t is used to label the corresponding edge.

Definition 3.17 (Marked Nets Equivalence). A pair of marked nets $a, b \in \mathcal{N}^\bullet$ are said to be in the relation $R_g \subseteq \mathcal{N}^\bullet \times \mathcal{N}^\bullet$ w.r.t. their reachability graphs, if there exists an edge-preserving graph isomorphism $f : \rho(a) \to \rho(b)$ between the two reachability graphs. Formally,

$$\begin{aligned} &\forall a, b \in \mathcal{N}^\bullet \,.\, (a, b) \in R_g \overset{\Delta}{\Longleftrightarrow} \\ &\quad \exists f : \rho(a) \to \rho(b) \,. \\ &\qquad \forall r \in \rho(b) \,.\, \exists s \in \rho(a) \,.\, f(s) = r \wedge \\ &\qquad \forall r, s \in \rho(a) \,.\, f(r) = f(s) \Rightarrow r = s \wedge \\ &\qquad \forall r, s \in \rho(a) \,.\, \forall t \in trs(a) \cup trs(b) \,.\, r \xrightarrow{t} s \Leftrightarrow f(r) \xrightarrow{t} f(s) \end{aligned} \tag{3.32}$$

The relation R_g is an equivalence relation and it is denoted by $\cdot \simeq_g \cdot$.

A Simple Interpreter

Petri Nets are generally used to model existing or desired systems and check their properties, for instance to know if a deadlock can occur during their execution. In such cases, one is concerned with all possible interleaved executions of concurrent entities, while the underlying scheduling strategy can be abstracted away. Conversely, when then language is used to model a new desired system, one may be interested in how a net is interpreted, for instance for simulating its behaviour. In this section an interpreter for simple nets is given.

Listing 3.1 PTNs interpreter for simple nets.

input: A net $a \in \mathcal{N}$
input: An initial state $q \in \mathcal{Q}$

NET-INTERPRETER(a, q)

```
1  if q ∉ markings(a) then
2      throw "illegal initial state"
3  end if
4  while enabled-set(a, q) ≠ ∅ do
5      if ∃t, h ∈ enabled-set(a, q) . λ(t) = λ(t) ∧ t ≠ h then
6          throw "ambiguous step"
7      end if
8      if ∃t ∈ enabled-set(a, q) . λ(t) = τ then
9          q ← fire(a, q, t)
       else
10         h ← WAIT-ACTION(a, enabled-set(a, q))
11         q ← fire(a, q, h)
12     end if
13 end while
14 return
```

Here, a net $a \in \mathcal{N}$ is said to be *simple* if and only if no reachable state $r \in \rho(a, q)$ enables two distinct transitions having the same action, including silent ones; i.e., for all $r \in \rho(a, q)$ and for all $t, h \in \textit{enabled-set}(a, r)$ if $\lambda(t) = \lambda(h)$ then $t = h$.

A simple interpreter for the PTNs language is given by the NET-INTERPRETER procedure in Lis. 3.1. Such procedure requires two arguments, a net $a \in \mathcal{N}$ and an initial state $q \in \mathcal{Q}$.

The interpreter throws an exception if the initial state is not valid (line 1) and it may also throw an exception if the net is not simple (line 5), and during the computation an ambiguous state is encountered. The interpreter runs until a final state is reached (line 4). When a silent transition is found (line 8), the interpreter selects it ignoring alternative actions. Otherwise, the interpreter waits that the environment selects one of the available actions through the WAIT-ACTION procedure (line 10). The WAIT-ACTION is considered a primitive procedure of the run-time support; hence, no implementation details are given here.

The selected action determines the transition to fire. It is not guaranteed that the interpreter is always reactive, i.e. it does not enter into a infinite loop, because it could exist an unlimited sequence of states, each one enabling a silent transition.

More sophisticated interpreters can be given but this is not the focus of the thesis. For example, one may allow multiple silent transitions at the same time, and consider those nets that are independent from a particular scheduling strategy.

3.2.4 Core Language Elements

In this section the PTNs bare language is enriched with special arcs and groups. Such constructs do not alter the language expressiveness, since they can be directly mapped to basic PTNs elements.

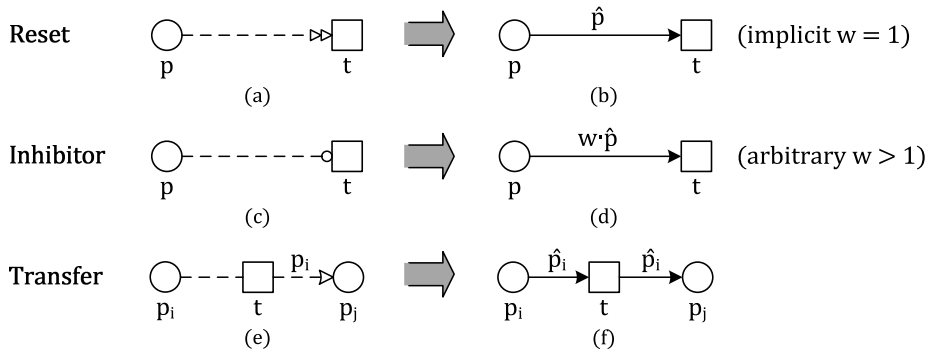


Fig. 3.6. The new elements added to the PTNs bare language. (a-b) A reset arc from p to t is a shorthand for a basic arc from p to t of instantaneous weight $\hat{p}$. (c-d) An inhibitor arc from p to t is a shorthand for a basic arc from p to t of instantaneous weight $w \cdot \hat{p}$ with $w > 1$. (e-f) A transfer arc from p_i to p_j passing through t is a shorthand for two basic arcs with the same instantaneous weight $\hat{p}_i$, one from p_i to t, the other from t to p_j.

The *reset arc* [60] construct is denoted as a dashed straight or curved line with a unfilled double arrow as in Fig. 3.6.a. It is always oriented from a place p to a

transition t, never the contrary. Its purpose is to empty the source place p every time the target transition t fires. The behavior of a reset arc is formally the same of a basic arc from p to t with instantaneous weight $\hat{p}$, as shown in Fig. 3.6.b.

The *inhibitor arc* [61] construct is denoted as a dashed straight or curved line with a unfilled circle at one end as in Fig. 3.6.c. It is always oriented from a place p to a transition t, never the contrary. Its purpose is to forbid the firing of the target transition t whenever the source place p is not empty. The behavior of an inhibitor arc is formally the same of a basic arc from p to t with instantaneous weight $2 \cdot \hat{p}$, as shows in Fig. 3.6.d. The choice of using 2 as a coefficient is not relevant: the same effect can be obtained with any weight greater than 1.

The *transfer arc* [62] construct is denoted as a dashed straight or curved line with a unfilled arrow at one end as shown in Fig. 3.6.e. A transfer arc connects a source place p_i with a target place p_j always passing through a transition t. The second part of the arc from t to p_j is annotated with the identifier p_i of the source place. The behavior of a transfer arc is formally the same of two related arcs one from the source place p_i to the intermediate transition t and the other from t to the target place p_j. The instantaneous weight $\hat{p}_i$ of both arcs is the number of tokens in the source place. The place annotation can be usually omitted, but it becomes mandatory when multiple transfer arcs pass through the same transition in order to correctly identify from which source place they come from, as happens in the net of Fig. 3.7.

A generic set of reset arcs or inhibitor arcs having the same target transition can be also denoted as a *reset group* or an *inhibitor group*, respectively. A reset group is depicted as a dashed closed line connected to the target transition with a reset arc, as shown in Fig. 3.8.a. The closed line encloses the source places that need to be reset when the target transition fires. An inhibitor group is denoted in similar way but using an inhibitor arc to connect the closed line with the target transition. The behavior of reset and inhibitor groups is formally defined in terms of reset and inhibitor arcs that in turn can be mapped to basic arcs of the PTNs bare language. For instance, the net in Fig. 3.8.a can be translated without efforts to the net in Fig. 3.8.b. In the given marking $\langle 1p_1 + 2p_2 + 3p_3 \rangle$, the transition t_1 turns out to be enabled because p_1 contains at least 1 token and both places p_5 and p_6 are empty. Places p_2, p_3 and p_4 in the reset group do not actually need to be considered in the activation of t_1 because they always contains the exact number of tokens required by their outgoing reset arcs. When t_1 fires, all tokens in the reset group are removed together with the token in p_1 reaching the final marking $\langle 1p_7 \rangle$.

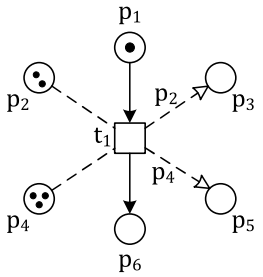


Fig. 3.7. A net with multiple transfer arcs passing through the same transition.

The following example shows how these new arcs can be used to define a net without the explicit use of place references. It also shows how a combination of special arcs can be used to copy the content of a place to another one.

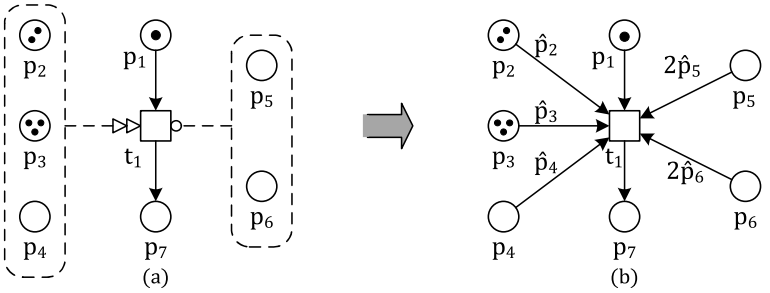


Fig. 3.8. (a) A simple net with a reset and a inhibitor group. (b) The same net translated in the PTNs bare language without special arcs.

Example 3.18. This example shows the use of all the three special arcs introduced by the PTNs core language, in particular it shows how two transfer arcs can be combined to implement a copy from one place to the other. The net in Fig. 3.9.b implements the steps given by the iterative algorithm in Fig. 3.9.a that in turn computes the simple function $z = x^y$ for any $x \in \mathbb{N}$ and $y \in \mathbb{N} \cup \{0\}$ using only basic arithmetical operators.

$z \leftarrow \textsc{Pow}(x, y)$

1 $z \leftarrow 1$
2 **while** $y > 0$ **do**
3 $\quad y \leftarrow y - 1$
4 $\quad z \leftarrow \sum_{i=1}^{x} z$
5 **end while**
6 **return** z

(a)

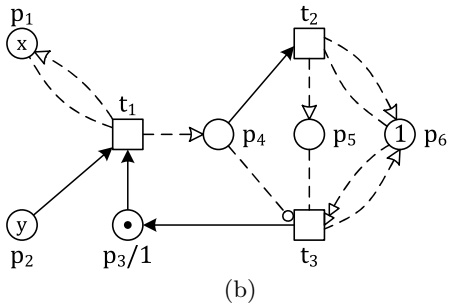


Fig. 3.9. (a) An iterative algorithm to compute the function x^y. (b) A net with special arcs that computes the same function following the logic of the given algorithm.

For any fixed number of tokens $x \in \mathbb{N}$ in p_1, the behavior of the net in Fig. 3.9.b can be described as follows. If $y = 0$ then the initial marking is also a final marking and the result 1 can be read from the last place p_6, indeed $\forall x \in \mathbb{N} \,.\, x^0 = 1$. If $y > 0$ then only t_1 is enabled: when it fires it consumes one token from p_2, one from p_3 and it copies x tokens in p_4. In the resulting configuration neither t_1 can fire because p_3 is empty, nor t_3 because p_4 contains some tokens.

At this stage only t_2 is enabled and can fire exactly x times copying the content of p_6 in p_5 until p_4 becomes empty. Now p_5 contains x times the content of p_6 and t_3 is enabled because both p_3 and p_4 are empty. When t_3 fires it removes all tokens from p_6, it moves in it the content of p_5 and it produces a token in p_3. The token in p_3 disables t_3 due to the imposed capacity: if p_3 had no such capacity limit, the transition t_3 could be fired indefinitely, altering the correct sequence of events. With a new token in p_3, the described computation can start again from t_1 as long as p_2 is not empty, but this time with a different value in p_6. A complete run of the net for $x = 2$ and $y = 3$ is represented in tabular form in Fig. 3.10 where the first row is the initial marking and each subsequent row reports the fired transition followed by the obtained marking. □

	p_1	p_2	p_3	p_4	p_5	p_6
	2	3	1	0	0	1
t_1	2	2	0	2	-	-
t_2	-	-	-	1	1	1
t_2	-	-	-	0	2	1
t_3	-	-	1	0	0	2
t_1	2	1	0	2	-	-
t_2	-	-	-	1	2	2
t_2	-	-	-	0	4	2
t_3	-	-	1	0	0	4
t_1	2	0	0	2	-	-
t_2	-	-	-	1	4	4
t_2	-	-	-	0	8	4
t_3	-	-	1	0	0	8

Fig. 3.10. A complete run of the net given in Fig. 3.9.b with $x = 2$ and $y = 3$.

As happens for the PTNs bare language, multi-arcs are not supported by the PTNs core language: special arcs cannot connect a transition with a place in its pre-set. In some cases such connections have no sense, e.g. in Fig. 3.11.a the transition t_{j1} can fire only if p_{i1} contains at least one token and is empty at the same time. In other cases multi-arcs may be useful, e.g. in the fragment of Fig. 3.11.b t_{j1} can fire whenever $x \geq w$ leading to a state in which p_{i1} is empty. The same effect can be obtained adding some constructs as exemplified in Fig. 3.11.c: a new transition t_{n+1} that fires immediately after t_{j1} can withdraw the tokens left in p_{i1}.

Particular attention must be paid to preserve the semantics of reset arcs. For instance, if there exists another transition t_{j2} that contains p_{i1} in its pre-set, t_{j2} is certainly disabled by the firing of t_{j1} in the case of Fig. 3.11.b, but not in the case of Fig. 3.11.c. Indeed, certain transitions need to be disabled until the reset is performed.

A generally applicable solution is exemplified in Fig. 3.11.d: for each transition t_{j2}, a new place p_{m+1+j2} with one token in it is added to the net; then, such place is connected with a double arc to the related transition, with an outgoing arc to the original transition t_{j1} and with an incoming arc from the new resetting transition t_{n+1}. The described construction can be applied whenever necessary to offer a multi-arc support without altering the bare language.

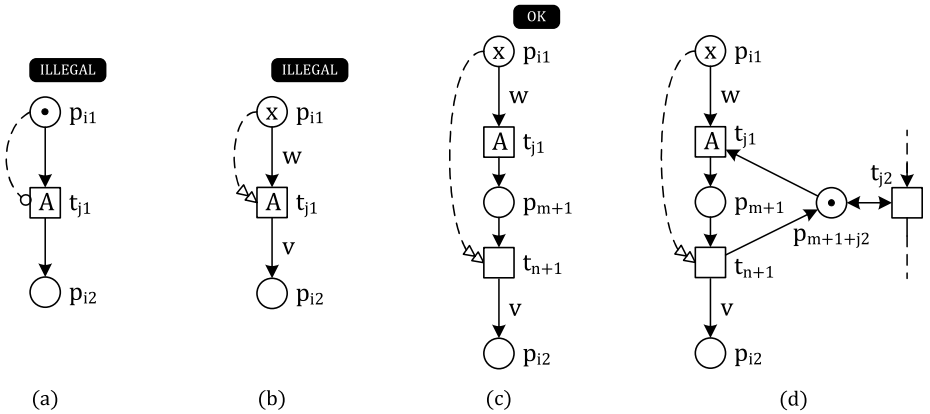


Fig. 3.11. (a) An inhibitor arc cannot connect a transition with a place in its pre-set. (b) The restriction also holds for reset arcs, but in this case such connection may be useful. (c) The reset effect can be emulated with an additional transition. (d) Conflicting transitions shall be disabled for not altering the original net semantics.

3.2.5 Petri Nets Classification

Nets modeled in the PTNs language can be classified as *self-modifying nets* with weighted arcs. The self-modifying nets language has been originally proposed by Valk in [55]. PTNs can also be seen as a restriction of the *generalized self-modifying nets* proposed in [57] by Dufourd et al. A generalized self-modifying net with m places $\{p_i\}_{i=1}^{m}$ is a Petri net in which the arc weights can be a generic finite polynomial of the form $\sum_{j=1}^{h} v_j \cdot \hat{x}_j^{e_j}$ where $h \in \mathbb{N}$, and for all $j \in [1, h]$ the coefficients $v_j, e_j \in \mathbb{N} \cup \{0\}$ and the variable $x_j \in \{p_i\}_{i=1}^{m}$.

In a net $a \in \mathcal{N}$, an instantaneous weight $w \cdot \hat{p}$ represents a polynomial of degree 1 in the form $v \cdot \hat{x} + k$, where $x \in pls(a)$ and $v, k \in \mathbb{N} \cup \{0\}$ are non-negative coefficients, such that $v + k > 0$ and $v \cdot k = 0$. As a consequence, the weight of an existing arc cannot be zero unless $\hat{p} = 0$ and it can only be in two forms $v \cdot \hat{x}$ or k, both representable as $w \cdot \hat{p}$ using the ideal place θ.

The remaining classes of Petri nets considered in this section are defined as restrictions of PTNs. This classification does not claim to be exhaustive; in particular, it does not include condition/event systems, elementary net systems, and other Petri nets with a step semantics [51] that are, for several reasons, less frequently used in comparison with Petri nets languages with an interleaving semantics, like PTNs. The latter ones are more studied from a theoretical point of view and more adopted in practical applications because they produce more compact models, thanks to special arcs and place counters. In this way one can capture the behavior of systems having an infinite state space with a finite model [58].

Standard Nets and Initial Markings

The definition of a net in PTNs does not include an explicit initial state, since it is not strictly necessary: a single transition with an empty pre-set can produce as many tokens as needed, then it can be disabled forever.

Seen from a different perspective, any net $a \in \mathcal{N}$ with $m \in \mathbb{N}$ places $pls(a) = \{p_i\}_{i=1}^{m}$ and an initial marking $q \triangleq \{p_i \mapsto v_i\}_{i=1}^{m}$ can be transformed into an equivalent net $b \in \mathcal{N}$ without a marking, by adding one transition t_{n+1}, one place p_{m+1}, and at most as many arcs as the non-empty places in the original net. The construction is depicted in Fig. 3.12 where an arc $\langle t_{n+1}, p_i, v_i, \theta \rangle$ is present only if $v_i > 0$. Formally, the b net can be constructed using the function in Eq. 3.33:

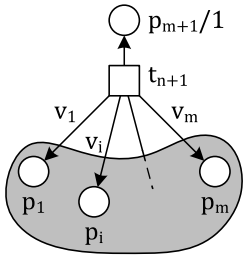


Fig. 3.12. Initial marking generic construction.

$$
\begin{aligned}
&unmark : \mathcal{N} \times \mathcal{Q} \to \mathcal{N} \ \ is \\
&\quad \forall a \in \mathcal{N} \,.\, \forall q \in \mathcal{Q} \,.\, \forall b \in \mathcal{N} \,.\, b = unmark(a, q) \iff \\
&\qquad places(b) = places(a) \cup \{\langle p_{m+1}, 1 \rangle\} \\
&\qquad transitions(b) = transitions(a) \cup \{\langle t_{n+1}, \tau \rangle\} \\
&\qquad arcs(b) = arcs(a) \cup \{\langle t_{n+1}, p_{m+1}, 1, \theta \rangle\} \cup \left(\bigcup_{p \in marked(q)} \{\langle t_{n+1}, p, q(p), \theta \rangle\} \right)
\end{aligned}
\tag{3.33}
$$

Definition 3.19 (Standard Net). A net is said to be *standard* [52] if all its transitions have at least an incoming arc of constant weight. This concept is captured by the class of nets $\mathcal{N}_{std}$ defined as follows:

$$
\begin{aligned}
&\forall a \in \mathcal{N} \,.\, a \in \mathcal{N}_{std} \iff \\
&\qquad \forall t \in trs(a) \,.\, pre\text{-}set(a, t) \neq \varnothing \ \wedge \ \exists w \in \mathbb{N} \,.\, (p, t, w, \theta) \in arcs(a)
\end{aligned}
\tag{3.34}
$$

On a standard net no transition can fire without tokens, hence an initial marking is mandatory in practice, otherwise the net has no chance to make progress. It is always possible to transform a marked net $a \in \mathcal{N}^{\bullet}$ to an equivalent marked net $b \in \mathcal{N}^{\bullet}$ with an initial marking of only one token. The construction is similar to the one presented in Eq. 3.33 and exemplified in Fig. 3.12: essentially, the arc $\langle t_{n+1}, p_{m+1} \rangle$ should be reversed and the capacity of p_{m+1} should be removed. Then, $q_s \triangleq \{p_{m+1} \mapsto 1\}$ can be taken as the default initial marking.

At the beginning, t_{n+1} is the only enabled transition in b, the state reached when t_{n+1} fires is exactly the initial marking of a with an additional empty place p_{m+1}, namely

$$
fire(b, t_{n+1}, q_s) = zero(net(b)) + initial\text{-}marking(a) \tag{3.35}
$$

The transition t_{n+1} fires one and only one time, from here b has the same behavior of a.

Pure Nets and Self-Loops

Definition 3.20 (Self-Loop). A *self-loop* on a net $a \in \mathcal{N}$, is a pair of vertices $u, v \in \mathit{vertices}(a)$ mutually connected by two arcs of opposite orientation. The fragment in Fig. 3.13.a is an example of self-loop.

Definition 3.21 (Pure Nets). A net $a \in \mathcal{N}$ is *pure* [63] if it does not contain self-loops. This concept is captured by the class of nets $\mathcal{N}_{pure}$ defined as:

$$\forall a \in \mathcal{N} \,.\, a \in \mathcal{N}_{pure} \triangleq \forall t \in \mathit{trs}(a) \,.\, \mathit{pre\text{-}set}(a,t) \cap \mathit{post\text{-}set}(a,t) = \varnothing \tag{3.36}$$

It can be easily proven that for any net $a \in \mathcal{N}$, if $\mathit{pre\text{-}set}(a,t) \cap \mathit{post\text{-}set}(a,t) = \varnothing$ for all transitions $t \in \mathit{trs}(a)$, then for all places $p \in \mathit{pls}(a)$ it also holds $\mathit{pre\text{-}set}(a,p) \cap \mathit{post\text{-}set}(a,p) = \varnothing$. Indeed, if there exists a place $g \in \mathit{pls}(a)$ such that $\mathit{pre\text{-}set}(a,g) \cap \mathit{post\text{-}set}(a,g)$ is not empty, then there exists $t \in \mathit{trs}(a)$ such that $(t,g) \in \mathit{edges}(a)$ and $(g,t) \in \mathit{edges}(a)$, hence $\mathit{pre\text{-}set}(a,t) \cap \mathit{post\text{-}set}(a,t) \neq \varnothing$ which implies $a \notin \mathcal{N}_{pure}$.

Pure nets are interesting because some PTNs variants may differ on how self loops are considered [63]. Furthermore, pure nets have a simpler mathematical representation based on transition matrices [14].

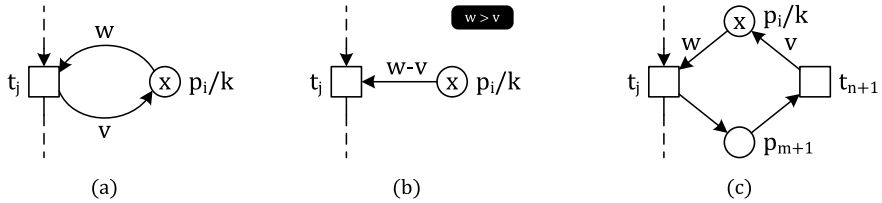


Fig. 3.13. (a) An example of self-loop (t_j, p_i). (b) Self-loop elimination through a single weighted arc assuming $w > v$. (c) Self-loop elimination through additional dummy constructs, useful when p_i is connected with other transitions and $w = v$.

A self-loop can be usually removed exploiting arc weights or adding some dummy constructs. For instance, consider the net in Fig. 3.13.a and assume $w > v$, the self-loop can be replaced by a single arc from p_i to t_j of weight $w - v$, as exemplified in Fig. 3.13.b. Conversely, if $w < v$ the solution is similar, but the arc has the opposite orientation and weight $v - w$.

This transformation does not work when the weights are equal or not constant. A solution for this case is exemplified in Fig. 3.13.c where new constructs are added. The exact behaviour of a self-loop is guaranteed if the transitions connected to p_i are disabled until t_{n+1} fires, restoring the initial value in p_i. This can be done using inhibitor arcs or similar constructions connected to p_{m+1}. In presence of instantaneous weights and places capacities, self-loops elimination is far from being an easy transformation.

Safe Nets and Place Capacities

Place capacity is a convenient construct but its presence does not alter the PTNs language expressiveness: every net $a \in \mathcal{N}$ can be transformed into an equivalent one without place capacities following the complementary place transformation [14]. In practice, every place p of limited capacity k is coupled with a new place h containing k tokens. A quantity of x tokens can be stored in p only if h contains at least x tokens when that happens.

Proposition 3.22 (Place Capacity Elimination). For every net $a \in \mathcal{N}$ and every initial marking $q \in \mathit{markings}(a)$, if a has $n \in \mathbb{N}$ places of limited capacity, there exists a net $b \in \mathcal{N}$ and an initial marking $r \in \mathit{markings}(b)$ such that b has the same behavior of a when executed from r, but b contains $n-1$ places with limited capacity.

$$\begin{aligned} &\forall a \in \mathcal{N} \,.\, \forall q \in \mathit{markings}(a) \,.\, \exists b \in \mathcal{N} \,.\, \exists r \in \mathit{markings}(b) \,. \\ &\quad (a, q) \simeq_g (b, r) \;\wedge\; \mathit{nlps}(b) < \mathit{nlps}(a) \end{aligned} \tag{3.37}$$

where $\mathit{nlps} : \mathcal{N} \to \mathbb{N} \cup \{0\}$ is the function that counts the number of non limited places, namely $\forall a \in \mathcal{N} \,.\, \mathit{nlps}(a) = |\{p \in \mathit{pls}(b) \mid \mathit{capacity}(b, p) < \omega\}|$.

Proof. The original net a has at least a place p_i with a limited capacity, otherwise there is nothing to do. The new net b is structurally identical to a except that: (1) the capacity limit of p_i is removed: $\mathit{capacity}(b, p_i) = \omega$, (2) a new place p_{m+1} is added, (3) b has a differen initial marking r that is like q except for the new place p_{m+1} initialized with $r(p_{m+1}) = \mathit{capacity}(a, p_i) - q(p_i)$ tokens; finally, (4) for every arc involving p_i and a transition t there exists a new arc with the same weight but different orientation that connects t and p_{m+1}. Given such construction, in every reachable state s the invariant $s(p_{m+1}) + s(p_i) = \mathit{capacity}(a, p_i)$ is preserved. □

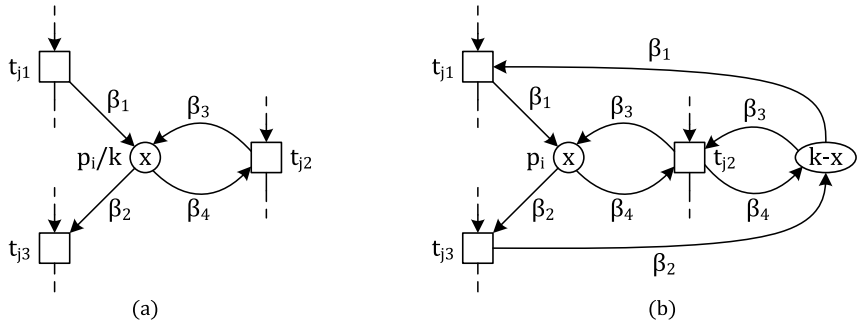


Fig. 3.14. (a) A net fragment containing a place p_i of limited capacity $k \in \mathbb{N}$ with $x \in \mathbb{N} \cup \{0\}$ initial tokens. (b) A fragment showing how the capacity k of p_i can be removed by adding a new place p_{m+1} and several new arcs.

Definition 3.23 (Safe Nets). A place $p \in pls(a)$ of a marked net $a \in \mathcal{N}^\bullet$ is *safe* [58] if and only if increasing its capacity does not alter the behavior of the net. An entire marked net is *safe* when all its places are safe, i.e. the given capacities do not constrain the behavior of the system, hence they can be removed. This class of nets is denoted by $\mathcal{N}^\bullet_{safe}$.

$$\forall a \in \mathcal{N}^\bullet \,.\, a \in \mathcal{N}^\bullet_{safe} \stackrel{\triangle}{\iff} a \simeq_g uncaps(a) \tag{3.38}$$

where $uncaps : \mathcal{N}^\bullet \to \mathcal{N}^\bullet$ removes all the place capacities of the specified net.

A net $a \in \mathcal{N}$ is said to be *structurally safe* if and only if it is safe for every valid initial marking. The set of structurally safe nets is denoted by $\mathcal{N}_{safe}$ defined as:

$$\forall a \in \mathcal{N} \,.\, a \in \mathcal{N}_{safe} \stackrel{\triangle}{\iff} \forall q \in markings(a) \,.\, (a, q) \in \mathcal{N}^\bullet_{safe} \tag{3.39}$$

Nets with Special Arcs

Petri nets with special arcs are classified in *post nets*, *inhibitor nets*, *reset nets*, and *transfer nets*. Their formal definition is given below.

Definition 3.24 (Post Net). A net is a *post net* [55] if and only if all its transitions have incoming arcs of constant weight. The set of all post nets is denoted by $\mathcal{N}_{post} \subseteq \mathcal{N}$ and is defined as follows:

$$\begin{aligned} &\forall a \in \mathcal{N} \,.\, a \in \mathcal{N}_{post} \stackrel{\triangle}{\iff} \\ &\quad \forall (x, y, w, h) \in arcs(a) \,.\, x \in pls(a) \wedge y \in trs(a) \Rightarrow h = 0 \end{aligned} \tag{3.40}$$

Definition 3.25 (Inhibitor Net). A net $a \in \mathcal{N}$ is an *inhibitor net* [61] if and only if its arcs are only weighted arcs or inhibitor arcs. The set of all inhibitor nets is denoted by $\mathcal{N}_{inhibitor} \subseteq \mathcal{N}$ and defined as follows:

$$\begin{aligned} \forall a \in \mathcal{N} \,.\, a \in \mathcal{N}_{inhibitor} \stackrel{\triangle}{\iff} &\forall (x, y, w, h) \in arcs(a) \,. \\ &(x \in pls(a) \wedge y \in trs(a) \Rightarrow h = \theta \vee w > 1) \vee \\ &(x \in trs(a) \wedge y \in pls(a) \Rightarrow h = \theta) \end{aligned} \tag{3.41}$$

Definition 3.26 (Reset Net). A net $a \in \mathcal{N}$ is a *reset net* [60] if and only if its arcs are only weighted arcs or reset arcs. The set of all reset nets is denoted by $\mathcal{N}_{reset} \subseteq \mathcal{N}$ and defined as follows:

$$\begin{aligned} \forall a \in \mathcal{N} \,.\, a \in \mathcal{N}_{reset} \stackrel{\triangle}{\iff} &\forall (x, y, w, h) \in arcs(a) \,. \\ &(x \in pls(a) \wedge y \in trs(a) \Rightarrow h = \theta \vee w > 1) \vee \\ &(x \in trs(a) \wedge y \in pls(a) \Rightarrow h = \theta) \end{aligned} \tag{3.42}$$

Definition 3.27 (Transfer Net). A net $a \in \mathcal{N}$ is a *transfer net* [62] if and only if its arcs are only weighted arcs or transfer arcs. The set of all transfer nets is denoted by $\mathcal{N}_{transfer} \subseteq \mathcal{N}$ defined as follows:

$$\begin{aligned} &\forall a \in \mathcal{N} \,.\, a \in \mathcal{N}_{transfer} \stackrel{\triangle}{\iff} \forall (x, y, w, h) \in arcs(a) \,. \\ &(x \in pls(a) \wedge y \in trs(a) \Rightarrow h = \theta \vee (w > 1 \wedge \exists d \in pls(a) \,.\, (y, d, 1, x) \in arcs(a))) \vee \\ &(x \in trs(a) \wedge y \in pls(a) \Rightarrow h = \theta \vee (w = 1 \wedge \exists (h, x, 1, h) \in arcs(a))) \end{aligned} \tag{3.43}$$

These definitions may differ from the ones given in the Petri nets literature in terms of multi-arcs support and arc orientation. For instance, in [64] the authors define reset arcs as a relation between transitions and places, hence a reset arc points to a place and that place can be in the pre-set of the transition. Arc orientation is not really relevant, while special arcs connected to the preset can be simulated as explained in the previous sections.

Classic Nets

Petri nets without special arcs are said to be *classic* [14]. The most frequently used classic Petri nets are basic or ordinary nets, free choice nets, marked graphs and state machines. Their definition is given below.

Definition 3.28 (Classic Net). A net $a \in \mathcal{N}$ is *classic* [14] if and only if it has no special arcs. The set of all classic nets is denoted by $\mathcal{N}_{classic} \subseteq \mathcal{N}$, defined as:

$$\forall a \in \mathcal{N} \,.\, a \in \mathcal{N}_{classic} \overset{\triangle}{\iff} \forall (x, y, w, h) \in arcs(a) \,.\, h = \theta \tag{3.44}$$

Definition 3.29 (Basic or Ordinary Nets). A net $a \in \mathcal{N}$ is *basic* or *ordinary* [14] if and only if all its arcs have constant weight 1. The set of all ordinary nets is denoted by $\mathcal{N}_{basic} \subseteq \mathcal{N}$ and defined as follows:

$$\begin{aligned} \forall a \in \mathcal{N} \,.\, a \in \mathcal{N}_{basic} \overset{\triangle}{\iff} a \in \mathcal{N}_{classic} \wedge \\ \forall (x, y) \in edges(a) \,.\, weight(a, x, y) = 1 \end{aligned} \tag{3.45}$$

Definition 3.30 (Extended Free Choice Net). A net $a \in \mathcal{N}$ is *extended free choice* [14] if and only if the net is ordinary and for any pair of places $p, h \in pls(a)$ their post-sets can be only disjoint or equal:

$$\begin{aligned} \forall a \in \mathcal{N} \,.\, a \in \mathcal{N}_{efc} \overset{\triangle}{\iff} a \in \mathcal{N}_{basic} \wedge \forall p, h \in pls(a) \,. \\ post\text{-}set(a, p) \cap post\text{-}set(a, h) \neq \varnothing \Rightarrow post\text{-}set(a, p) = post\text{-}set(a, h) \end{aligned} \tag{3.46}$$

Definition 3.31 (Free Choice Net). A net $a \in \mathcal{N}$ is *free choice* [14] if and only if the net is ordinary and for any pair of places $p, h \in pls(a)$ their post-sets can be only equal or have the same size:

$$\begin{aligned} \forall a \in \mathcal{N} \,.\, a \in \mathcal{N}_{fc} \overset{\triangle}{\iff} a \in \mathcal{N}_{basic} \wedge \forall p, h \in pls(a) \,. \\ post\text{-}set(a, p) \cap post\text{-}set(a, h) \neq \varnothing \Rightarrow |post\text{-}set(a, p)| = |post\text{-}set(a, h)| \end{aligned} \tag{3.47}$$

Definition 3.32 (Marked Graph). A net $a \in \mathcal{N}$ is a *marked graph* [14] if and only if the net is ordinary and each place can have only a transition in its pre-set and only one transition in its post-set:

$$\begin{aligned} \forall a \in \mathcal{N} \,.\, a \in \mathcal{N}_{mg} \overset{\triangle}{\iff} a \in \mathcal{N}_{basic} \wedge \\ \forall p \in pls(a) \,.\, |pre\text{-}set(a, p)| = |post\text{-}set(a, p)| = 1 \end{aligned} \tag{3.48}$$

Definition 3.33 (State Machine). A net $a \in \mathcal{N}$ is a *state machine* [14] if and only if the net is ordinary and each transition can have only a place in its pre-set and only a place in its post-set:

$$\begin{aligned} \forall a \in \mathcal{N} \,.\, a \in \mathcal{N}_{sm} \overset{\triangle}{\iff} a \in \mathcal{N}_{basic} \wedge \\ \forall t \in trs(a) \,.\, |pre\text{-}set(a, t)| = |post\text{-}set(a, t)| = 1 \end{aligned} \tag{3.49}$$

3.3 Coloured Petri Nets

Coloured Petri Nets (CPNs) is a graphical language for the modeling and validation of concurrent and distributed systems. It extends classical Petri nets with inscriptions and expressions encoded into an high-level general-purpose programming language. Such language is called CPN ML programming language and is a a subset of the functional programming language Standard ML [65]. It supports data types, data manipulation and simplifies the construction of compact models.

CPNs is an industrial strength modeling language best suited for the representation of network protocols, concurrent systems, as well as business processes. The main goal of CPNs is to produce executable models that allow one to analyze the behavior of a concurrent system through simulation or by applying verification methods to check the presence of desired properties. All these operations are supported through a software tool called CPN Tools [66].

3.3.1 Graphical Elements and Syntax

The graphical elements of the CPNs language are summarized in Fig. 3.15, which provides all variants of each element representation. The position of each inscription is not arbitrary, but follows the CPNs conventions.

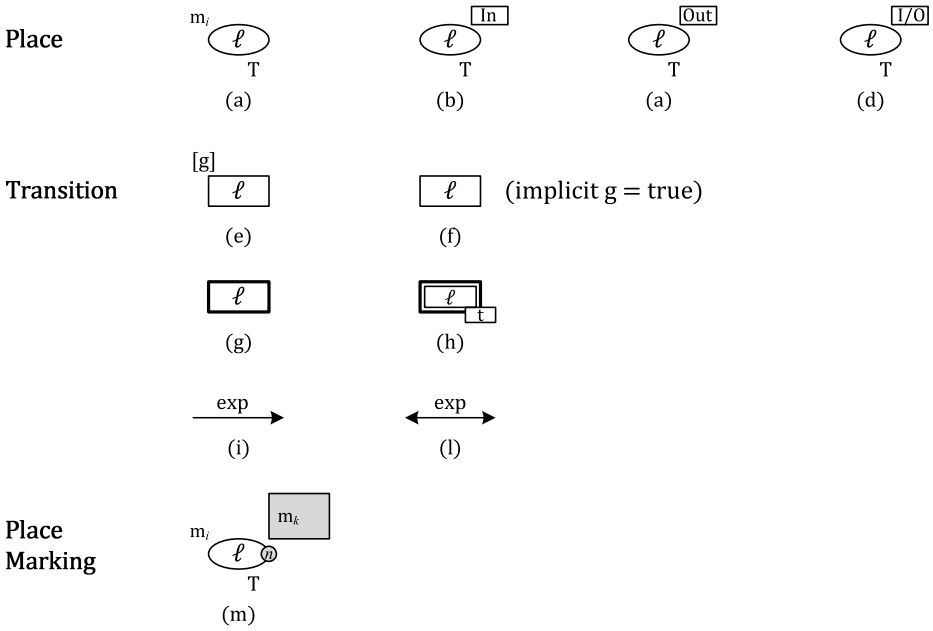


Fig. 3.15. CPN language elements

A *place* is depicted as an ellipse enriched with some annotations, as illustrated in Fig. 3.15.a. In particular, a descriptive label ℓ for the place is specified inside the ellipse, while the annotation T below it denotes the place type or colour set,

i.e. the set of data values that a token inside it can have; finally m_i is the initial, eventually empty, marking associated to the place. Places are used to represent the state of a system. For this purpose, each place can be marked with one or more tokens, as exemplified in Fig. 3.15.m where, besides the initial place marking m_i, the current (eventually empty) place marking m_k is specified, together with the total number n of contained tokens. A place marking is represented as a multiset of tokens each one characterized by a particular data value, called *token colour*.

A CPN model can be organized into a set of *modules* in a hierarchical way. A module can be defined from scratch or starting from existing ones, and it can be used in other context; anyway, it cannot be part of a recursive definition. A *port place* representing the interface of a module is annotated by positioning next to it a rectangular tag specifying whether the place is an input (In), output (Out), or input/output (I/O) port, as in Fig. 3.15.b, Fig. 3.15.c, and Fig. 3.15.d, respectively.

A *transition* is represented as a rectangular box containing a descriptive label ℓ and eventually annotated with a guard expression g, as illustrated in Fig. 3.15.e. The guard expression is written in square brackets and positioned next to the transition. It can be any boolean expression written in ML. An enabled transition is depicted as a thick square rectangle, as in Fig. 3.15.g. Finally, a substitution transition is represented as a rectangular box with a double-lines border annotated with a substitution tag t placed next to it, containing the name of the related substitution module, as exemplified in Fig. 3.15.h. A substitution transition cannot have a guard.

An *arc* between a place and a transition is annotated with an arc expression *exp*, as in Fig. 3.15.i. A double headed arc, as the one in Fig. 3.15.l, can be used as a shorthand for two oppositely directed arcs between the same place and transition which share also the same arch expression *exp*.

Notice that markings, arc expressions, and transition guards are usually encoded using the CPN ML language. However, as stated in [13], the CPNs language is independent from the chosen inscription language, and in the following the usual mathematical notation is adopted.

3.3.2 Language Interpretation

As done for the PTNs language, the informal semantics of the CPNs language can be described in terms of states and transitions. The *state* of the net is given by the distribution of tokens inside the places. However, in CPNs a token has an associated data value, called *colour*. In particular, each place in the net declares the set of colours it can contains and tokens inside it can have only a value compliant with such set. The content of a place is called *place marking* and is represented as a multiset, while the state of the net is usually called *marking*.

The expression associated to each arc determines when a transition is enabled in a given marking. More specifically, a transition is *enabled* when the expression of all incoming arcs can be successfully evaluated. An arc expression can be evaluated when all variables inside it can be bound to values of the correct type.

The execution of a transition removes tokens from its input places and adds tokens to the output places. The colours of tokens that are removed from the input places and added to the output places are determined by the corresponding arc

expressions. Transition *guards* can also be used to put an extra constraints for determining when a transition is enabled.

A *binding* is an assignment of values to all variables of a transition. In particular, the variables of a transition t are the free variables appearing in the guard of t, together with the variables in the arc expression of the arcs connected to t. A *binding element* is a pair composed of a transition and a binding for it.

A model is said to be *deterministic* if each reachable marking has exactly one enabled transition, with exactly one enabled binding; otherwise, the existence of a reachable marking with more than one enabled binding element makes a model *non-deterministic*. Notice that it is the choice between the enabled transitions that is non-deterministic, while the individual transition execution is deterministic. Two transitions are in *conflict* if both of them are enabled in a given marking but only one of them can occur, since each of them needs the same token colour in a particular place p and there is only one token of this colour in p. Conversely, two transitions can occur *concurrently* if they need disjoint set of input tokens, hence they can execute without competition or interference. Let us notice that besides the concurrency between two distinct transitions, a form of *self-concurrency* can occur when there is non-determinism on which coloured token has to be used: a place marking is implemented as a multiset and not as a queue, any token satisfying the arc expression can be consumed at a certain point.

A non-empty, finite multiset of concurrently enabled binding elements is called *step*. An interesting property of any CPNs model is that, given a set of concurrent binding elements, the effects of their concurrent execution is the same as the sum of the effects caused by their sequential execution. In other words, the model reaches the same final marking running them concurrently or sequentially by performing each binding element in any arbitrary order.

A CPNs model can be organized into a set of *modules*. This is an important feature for three main reasons: (1) it can become impractical to draw a large system with a single CPNs model, (2) designers need abstractions that make it possible to concentrate on only few details at time, and (3) a module can be defined once and used repeatedly in different context.

The interface of a module is determined by a set of places used for exchanging tokens with the external environment. These tokens are called *port places* and as explained in the previous section they can be recognized by the presence of a tag which specifies its input, output or mixed type.

In a hierarchical net a module is represented using a *substitution transition* which represents the compound behaviour of its corresponding submodule. The input and output places of a substitution transition are called *input* and *output sockets*, respectively. They represents the interface of the substitution transition and have to be properly related with the port places of the corresponding submodule. A substitution transition cannot have a guard.

The relations between the port places of a submodule and the sockets of a substitution transition is called *port-socket relation*. This relation implies that connected ports and sockets represent different views of the same places; therefore, they always have the same marking, besides to the same colour set. Moreover, if a port place does not have an initial marking, it obtains its initial marking form the related socket place. This mechanism can be used to obtain a sort of

parameterisation, since the initial marking of a socket place can be used to transfer parameters to a submodule.

An important consequence of this module decomposition is that the same module can be used as submodule of several substitution transitions, namely there can be different instances of the same module in a given model. Places and transitions in a certain module instance are referred to as *place instances* and *transition instances*, and each instance is characterized by its own marking.

The relationships between modules in a hierarchical model can be represented through a directed graph, where each node represents a module and each arc represents a substitution transition. The hierarchical graph is required to be acyclic: namely, it is not possible for a module to be submodule of itself. This constraint ensures that there is only a finite number of instances of each module when the model is instantiated. Moreover, it is required that the number of instances of each module is determined at design-time and it is not possible to instantiate new modules during the simulation.

Finally, another interesting characteristics of hierarchical CPNs is the possibility to glue together places of different modules into one compound place. These compound places are called *fusion sets* and they are similar to global variables: places in the same fusion set always share the same marking and they shall have identical colour sets and initial markings.

An important property of each hierarchical CPNs model is that it can be always *unfolded* into an equivalent non-hierarchical CPNs model with the same behaviour. This operation can be performed using the following steps: (1) each substitution transition is replaced with the content of its associated submodule, while the related ports and sockets are merged together, (2) the content of the root of all module hierarchies are collected into a single module, and (3) the places in a fusion set are merged into a unique place. A consequence of this property is that any system which can be represented with a hierarchical CPNs model, can also be represented using a non-hierarchical CPNs model; hence, CPNs modules do not increase the language expressiveness.

A similar property holds also between CPNs models and low-level Petri nets: every non-hierarchical CPNs model can be transformed into an equivalent, eventually infinite, Petri net [63]. The idea is that each CPNs place is replaced with as many places as there are colours in the CPNs place type. Each CPNs transition is replaced with as many transitions as there are possible bindings satisfying the guard for the CPNs transition. It follows that for CPNs models with infinite colour sets, the corresponding Petri net has an infinite number of places and transitions.

The following example illustrates how arc expressions can be used to reduce the number of needed control-flow relations and to simplify the overall net construction. Such example regards the realization of a network packet switch.

Example 3.34 (Network packet switch). The CPNs model in Fig. 3.16 switches the received data value to one of its output places on the basis of a condition evaluation. This condition regards the comparison between a control value received in input and a constant parameter. The model has three input places *par*, *in* and *data*, and two output places out_1 and out_2: if the value read from *in* is less than the value read from *par*, then the data value read from *data* is placed into out_1, else it is

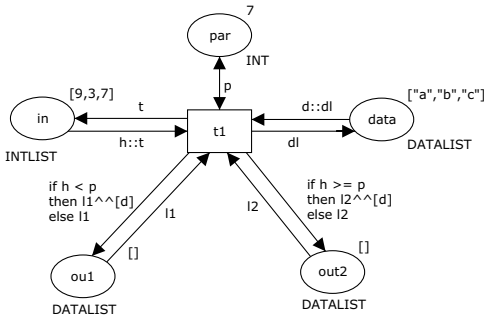


Fig. 3.16. Example of if operator with parameter in CPN.

placed into out_2. Places *in*, *data*, out_1 and out_2 contain a single token representing a list of values, while *par* contains a single integer value equals to 7.

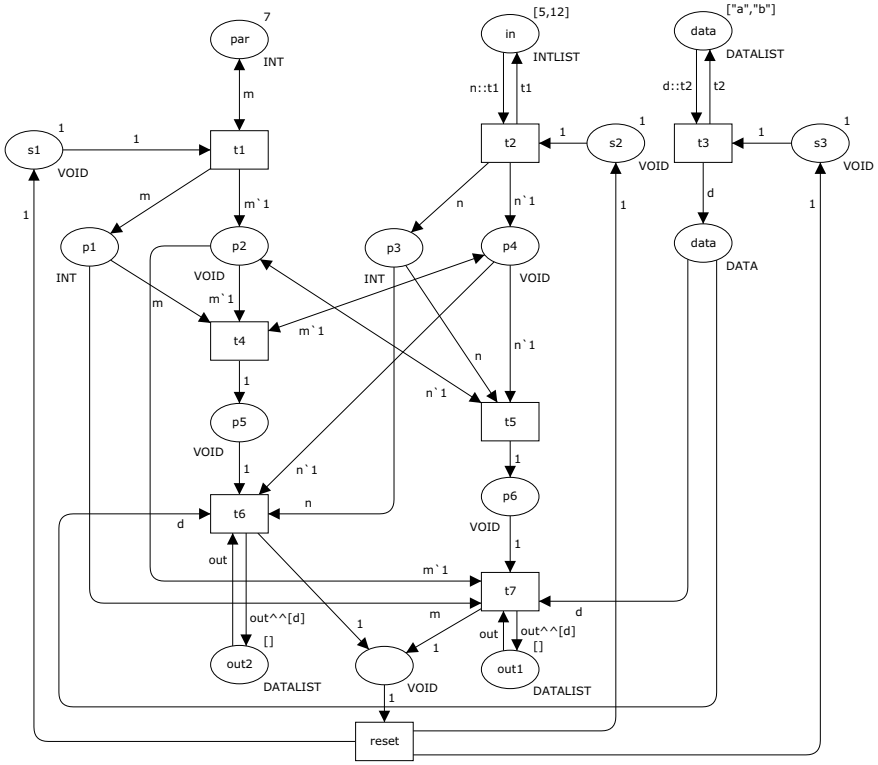


Fig. 3.17. Example of if operator with parameter in CPN.

The use of a single token containing a list, whose value is updated by the execution of each transition, ensures that its elements are consumed respecting the insertion order. Indeed, each transition is connected with a double arc to those

places, if the transition has to read a value from a place, it reads the list and removes its head element by storing back the list tail; conversely, if the transition has to write on a place, it reads the list and appends the new element at its end. For instance, transition t_2 reads the list $n :: t_1$ from in, where n is the list head, and writes back the list tail t_1; similarly, transition t_7 reads the list out from out_1 and writes back the list obtained by concatenating out and the element d, denoted as $out^\frown[d]$. It classical multiset of tokens are used in place of lists, the consumption order does not necessarily reflect the production order.

At the beginning transition t_1 is enabled, it reads the control value t, the parameter p, and the data value d from its input places. If $t < p$, then d is placed in out_1, otherwise it is placed in out_2.

The net in Fig. 3.17 performs the same operation but using control-flow relations in place of high-level arc expressions. Initially, transitions t_1, t_2 and t_3 are enabled, transition t_1 inserts into p_1 the value m read from place par and into p_2 m void tokens. Similarly, transition t_2 puts into p_3 the input control value n read from in_1, and n void tokens into p_4 . If $n < m$, then transition t_5 is enabled, which reads the number of tokens to consume from p_3, and the corresponding number of tokens from p_2 and p_4. Finally, transition t_7 reads the data token and puts it into out_1. Otherwise, transition t_4 is enabled which performs a symmetric operation by consuming m tokens from p_2 and p_4. Finally, transition t_6 reads the data token and puts it into out_2. The last transition $reset$ resets the net and enables another execution by placing a token into s_1, s_2 and s_3. These tokens ensure that another input is read only when the structure is completely clean. □

3.3.3 Formal Semantics

A complete formalization of the CPNs language in terms of mathematical functions can be found in [13]. This section summarizes its main semantical aspects following the approach adopted in Sec. 3.2.3. Since the CPNs definition is independent from the adopted concrete inscription language [13], this introduction can abstract from the CPN ML language which is replaced by the usual mathematical notation for expressing markings, transition guards, and arc expressions. In particular, the notion of multiset, graph, and their related operations are taken from Chap. 2. Moreover, in the following $\mathcal{E} \subseteq \mathcal{U}$ denotes the set of available expressions, $type : \mathcal{E} \to \mathcal{U}$ denotes the type of an expression $e \in \mathcal{E}$, and $var : \mathcal{E} \to \mathcal{U}$ denotes the set of *free variables* in an expression $e \in \mathcal{E}$. The set of all expressions $e \in \mathcal{E}$ such that $var(e) \subseteq V$ for a certain $V \subseteq \mathcal{U}$ is denoted as $\mathcal{E}(V)$.

Definition 3.35 (Coloured Petri Nets). A *non-hierarchical coloured Petri net* is a tuple $\langle V, E, Y, B, C, D, F, G, A, I, L \rangle$ that belongs to the set $\mathcal{C}$ defined as follows:

$$\begin{aligned} \forall a \in \mathcal{U} \,.\, a \in \mathcal{C} \stackrel{\Delta}{\Longleftrightarrow} & \\ & a = \langle V, E, Y, B, C, D, F, G, A, I, L \rangle \quad \wedge \\ & \langle V, E \rangle \in \mathcal{G} \quad \wedge \quad Y : V \to \{\mathsf{PL}, \mathsf{TR}\} \quad \wedge \\ & F : P \to C \quad \wedge \quad G : T \to \mathcal{E}(V) \quad \wedge \\ & A : E \to \mathcal{E}(V) \quad \wedge \quad I : P \to \mathcal{E}(\varnothing) \end{aligned} \tag{3.50}$$

such that:

- $\langle V, E \rangle \in \mathcal{G}$ is a finite directed graph.
- $Y : V \to \{\textsf{PL}, \textsf{TR}\}$ is a total function that associates to each vertex its type, where **PL** stands for place and **TR** for transition. The function Y induces a partition of V composed of two disjoint sets P and T, such that $P = \{v \in V \mid Y(v) = \textsf{PL}\}$ is the set of all places, $T = \{v \in V \mid Y(v) = \textsf{TR}\}$ is the set of all transitions, $V = P \cup T$ and $P \cap T = \varnothing$.
- B is a finite set of labels for places and transitions.
- C is a finite set of non-empty colour sets.
- D is a finite set of typed variables such that $\forall v \in V \,.\, \mathit{type}(v) \in C$.
- $F : P \to C$ is a *colour set function* that assigns a colour set to each place.
- $G : T \to \mathcal{E}(V)$ is a *guard function* that assigns a guard expression to each transition t such that $\forall t \in T \,.\, \mathit{type}(G(t)) = \mathbb{B}$.
- $A : E \to \mathcal{E}(V)$ is an *arc expression function* that assigns an arc expression to each arc $e \in E$, such that the type of each expression is a multiset whose values belong to the colour set of the place p connected to e:

$$\forall e \in E \,.\, (e = (p, v) \vee e = (u, p)) \wedge p \in P \,.\, \mathit{elements}(\mathit{type}(A(e))) \subseteq F(p)$$

- $I : P \to \mathcal{E}(\varnothing)$ is an *initialization function* that assigns an initialization expression to each place p, such that $\mathit{elements}(\mathit{type}(I(p))) \subseteq F(p)$. Notice that the initialization expression must be a closed expression: namely, it cannot have free variables.
- $L : V \to B$ is a label function that associates to each place or transition in the net a unique label.

The basic access functions are summarized in Tab. 3.5 and they are defined as follows: for any $a \in \mathcal{C}$ such that $a = \langle V, E, Y, B, C, D, F, G, A, I, L \rangle$, $\mathit{vertices}(a) = V$, $\mathit{edges}(a) = E$, $\mathit{places}(a) = \{v \in V \mid Y(v) = \textsf{PL}\}$, and $\mathit{transitions}(a) = \{v \in V \mid Y(v) = \textsf{TR}\}$, while $\mathit{colour\text{-}sets}(a) = C$, and $\mathit{variables}(a) = D$.

Table 3.5. Basic operations on CPNs nets.

Symbol	Function	Description	Ref
	$\mathit{vertices} : \mathcal{C} \to \wp(\mathcal{U})$	Vertices of the underlying graph.	*inline*
	$\mathit{edges} : \mathcal{C} \to \wp(\mathcal{U})$	Edges of the underlying graph.	*inline*
	$\mathit{places} : \mathcal{C} \to \wp(\mathcal{U})$	Vertices that are places.	*inline*
	$\mathit{transitions} : \mathcal{C} \to \wp(\mathcal{U})$	Vertices that are transitions.	*inline*
	$\mathit{colour\text{-}sets} : \mathcal{C} \to \wp(\mathcal{U})$	Set of non-empty colour sets.	*inline*
	$\mathit{variables} : \mathcal{C} \to \wp(\mathcal{U})$	Set of typed variables.	*inline*
	$\mathit{colour\text{-}set} : \mathcal{C} \times \mathcal{U} \rightharpoonup \mathcal{U}$	Place colour.	Eq. 3.51
	$\mathit{guard} : \mathcal{C} \times \mathcal{U} \rightharpoonup \mathcal{E}$	Transition guard.	Eq. 3.52
	$\mathit{arc\text{-}expr} : \mathcal{C} \times \mathcal{U} \times \mathcal{U} \rightharpoonup \mathcal{E}$	Arc expression.	Eq. 3.53
	$\mathit{init} : \mathcal{C} \times \mathcal{U} \rightharpoonup \mathcal{U}$	Place initialization.	Eq. 3.54
	$\mathit{label} : \mathcal{C} \times \mathcal{U} \rightharpoonup \mathcal{U}$	Vertex label.	Eq. 3.55

The colour set function $\mathit{colour\text{-}set} : \mathcal{C} \times \mathcal{U} \rightharpoonup \mathcal{U}$ is defined in Eq. 3.51: it returns the colour set associated to a particular place.

$$\begin{aligned} &\mathit{colour\text{-}set} : \mathcal{C} \times \mathcal{U} \rightharpoonup \mathcal{U} \ \mathit{is} \\ &\quad \forall a \in \mathcal{C} \,.\, \forall u \in \mathcal{U} \,. \\ &\qquad \mathit{colour\text{-}set}(a, u) = \begin{cases} F(u) & \text{if } u \in \mathit{places}(a) \\ \uparrow & \text{otherwise} \end{cases} \end{aligned} \tag{3.51}$$

The guard function $\mathit{guard} : \mathcal{C} \times \mathcal{U} \rightharpoonup \mathcal{E}$ is defined in Eq. 3.52: it associates to each transition a guard expression of boolean type.

$$\begin{aligned} &\mathit{guard} : \mathcal{C} \times \mathcal{U} \rightharpoonup \mathcal{E} \ \mathit{is} \\ &\quad \forall a \in \mathcal{C} \,.\, \forall u \in \mathcal{U} \,. \\ &\qquad \mathit{guard}(a, u) = \begin{cases} G(u) & \text{if } u \in \mathit{transitions}(a) \\ \uparrow & \text{otherwise} \end{cases} \end{aligned} \tag{3.52}$$

The arc expression function $\mathit{arc\text{-}expr} : \mathcal{C} \times \mathcal{U} \times \mathcal{U} \rightharpoonup \mathcal{E}$ associates to each arc in the net an expression as illustrated in Eq. 3.53.

$$\begin{aligned} &\mathit{arc\text{-}expr} : \mathcal{C} \times \mathcal{U} \times \mathcal{U} \rightharpoonup \mathcal{E} \ \mathit{is} \\ &\quad \forall a \in \mathcal{C} \,.\, \forall u, v \in \mathcal{U} \,. \\ &\qquad \mathit{arc\text{-}expr}(a, u, v) = \begin{cases} A(u, v) & \text{if } (u, v) \in \mathit{edges}(a) \\ \uparrow & \text{otherwise} \end{cases} \end{aligned} \tag{3.53}$$

The initialization function $\mathit{init} : \mathcal{C} \times \mathcal{U} \rightharpoonup \mathcal{U}$ assigns an initialization expression to each place as explained in Eq. 3.54.

$$\begin{aligned} &\mathit{init} : \mathcal{C} \times \mathcal{U} \rightharpoonup \mathcal{U} \ \mathit{is} \\ &\quad \forall a \in \mathcal{C} \,.\, \forall u \in \mathcal{U} \,.\, a = \langle V, E, Y, B, C, D, F, G, A, I, L \rangle \,. \\ &\qquad \mathit{init}(a, u) = \begin{cases} I(u) & \text{if } u \in \mathit{places}(a) \\ \uparrow & \text{otherwise} \end{cases} \end{aligned} \tag{3.54}$$

Finally, the label function $\mathit{label} : \mathcal{C} \times \mathcal{U} \rightharpoonup \mathcal{U}$ assigns to each vertex in the net a unique label. It is defined in Eq. 3.55.

$$\begin{aligned} &\mathit{label} : \mathcal{C} \times \mathcal{U} \rightharpoonup \mathcal{U} \ \mathit{is} \\ &\quad \forall a \in \mathcal{C} \,.\, \forall u \in \mathcal{U} \,.\, a = \langle V, E, Y, B, C, D, F, G, A, I, L \rangle \,. \\ &\qquad \mathit{label}(a, u) = \begin{cases} L(u) & \text{if } u \in \mathit{vertices}(a) \\ \uparrow & \text{otherwise} \end{cases} \end{aligned} \tag{3.55}$$

Definition 3.36 (CPNs Module). A CPNs *module* is a tuple $\langle C, T_{sub}, P_{port}, PT \rangle$ that belongs to the set $\mathcal{C}_{\mathcal{M}}$ defined as follows:

$$\begin{aligned} &\forall a \in \mathcal{U} \,.\, a \in \mathcal{C}_{\mathcal{M}} \overset{\Delta}{\Longleftrightarrow} \\ &\quad a = \langle b, T_{sub}, P_{port}, PT \rangle \quad \wedge \\ &\quad b \in \mathcal{C} \quad \wedge \quad T_{sub} \subseteq \mathit{transitions}(C) \quad \wedge \\ &\quad P_{port} \subseteq \mathit{places}(C) \quad \wedge \quad PT : P_{port} \rightarrow \{\mathsf{IN}, \mathsf{OUT}, \mathsf{I/O}\} \end{aligned} \tag{3.56}$$

such that:

- $b \in \mathcal{C}$ is a non-hierarchical coloured petri net.
- $T_{sub} \subseteq \mathit{transitions}(b)$ is the set of substitution transitions.
- $P_{port} \subseteq \mathit{places}(b)$ is the set of port places.
- $PT : P_{port} \to \{\mathsf{IN}, \mathsf{OUT}, \mathsf{I/O}\}$ is a port type function that assigns to each port place its type.

Table 3.6. Basic operations on CPNs module.

Symbol	Function	Description	Ref
	$\mathit{net} : \mathcal{C}_{\mathcal{M}} \to \mathcal{C}$	The contained coloured Petri net.	*inline*
	$\mathit{sub\text{-}transitions} : \mathcal{C}_{\mathcal{M}} \rightharpoonup \mathcal{U}$	Set of substitution transitions.	*inline*
	$\mathit{port\text{-}places} : \mathcal{C}_{\mathcal{M}} \rightharpoonup \mathcal{U}$	Set of port places.	*inline*
	$\mathit{port\text{-}type} : \mathcal{C}_{\mathcal{M}} \times \mathcal{U} \rightharpoonup \mathcal{U}$	Type of a port place.	*inline*
	$\mathit{is\text{-}port} : \mathcal{C}_{\mathcal{M}} \times \mathcal{U} \rightharpoonup \mathbb{B}$	True if the place is a module port, false otherwise.	Eq. 3.57
	$\mathit{is\text{-}substitution} : \mathcal{C}_{\mathcal{M}} \times \mathcal{U} \rightharpoonup \mathbb{B}$	True if the transition is a substitution transition, false otherwise.	Eq. 3.58
	$\mathit{socket\text{-}type} : \mathcal{C}_{\mathcal{M}} \times \mathcal{U} \times \mathcal{U} \rightharpoonup \mathcal{U}$	It returns the type of a socket place for a substitution transition	Eq. 3.59
	$\mathit{trans\text{-}sockets} : \mathcal{C}_{\mathcal{M}} \times \mathcal{U} \rightharpoonup \wp(\mathcal{U})$	It returns the socket places of a substitution transition	Eq. 3.60

The basic access functions are summarized in table Tab. 3.6 and they are defined as follows. For any CPNs module $a \in \mathcal{C}_{\mathcal{M}}$ such that $a = \langle b, T_{sub}, P_{port}, PT \rangle$, $\mathit{net}(a) = b$, $\mathit{sub\text{-}transitions}(a) = T_{sub}$, $\mathit{port\text{-}places}(a) = P_{port}$, and $\mathit{port\text{-}type}(a, u) = PT(u)$ if $u \in \mathit{port\text{-}places}(a)$, or $\uparrow$ otherwise.

The function $\mathit{is\text{-}port} : \mathcal{C}_{\mathcal{M}} \times \mathcal{U} \rightharpoonup \mathbb{B}$ determines if a place is a port for a CPNs module or not, its behaviour is defined in Eq. 3.57.

$$\begin{aligned} &\mathit{is\text{-}port} : \mathcal{C}_{\mathcal{M}} \times \mathcal{U} \rightharpoonup \mathbb{B} \ \mathit{is} \\ &\quad \forall a \in \mathcal{C}_{\mathcal{M}} \,.\, \forall u \in \mathcal{U} \,. \\ &\qquad \mathit{is\text{-}port}(a, u) = \begin{cases} \mathsf{T} & \text{if } u \in \mathit{places}(a) \wedge u \in \mathit{port\text{-}places}(a) \\ \mathsf{F} & \text{if } u \in \mathit{places}(a) \wedge u \notin \mathit{port\text{-}places}(a) \\ \uparrow & \text{otherwise} \end{cases} \end{aligned} \tag{3.57}$$

Function $\mathit{is\text{-}substitution} : \mathcal{C}_{\mathcal{M}} \times \mathcal{U} \rightharpoonup \mathbb{B}$ returns true if the given transition is a substitution transition, false otherwise. Its definition is reported in Eq. 3.58.

$$\begin{aligned} &\mathit{is\text{-}substitution} : \mathcal{C}_{\mathcal{M}} \times \mathcal{U} \rightharpoonup \mathbb{B} \ \mathit{is} \\ &\quad \forall a \in \mathcal{C}_{\mathcal{M}} \,.\, \forall u \in \mathcal{U} \,. \\ &\qquad \mathit{is\text{-}substitution}(a, u) = \begin{cases} \mathsf{T} & \text{if } u \in \mathit{transitions}(a) \wedge u \in \mathit{sub\text{-}transitions}(a) \\ \mathsf{F} & \text{if } u \in \mathit{transitions}(a) \wedge u \notin \mathit{sub\text{-}transitions}(a) \\ \uparrow & \text{otherwise} \end{cases} \end{aligned} \tag{3.58}$$

The function $\textit{socket-type} : \mathcal{C}_{\mathcal{M}} \times \mathcal{U} \times \mathcal{U} \nrightarrow \mathcal{U}$ returns the type of a socket place for a substitution transition. Its definition in given in Eq. 3.59.

$$\begin{aligned} &\textit{socket-type} : \mathcal{C}_{\mathcal{M}} \times \mathcal{U} \times \mathcal{U} \nrightarrow \mathcal{U} \ \textit{is} \\ &\quad \forall a \in \mathcal{C}_{\mathcal{M}} \,.\, \forall u, v \in \mathcal{U} \,.\, \\ &\qquad \textit{socket-type}(a,u,v) = \begin{cases} \textit{port-type}(a,v) & \text{if } \textit{is-substitution}(a,u) \wedge \\ & \quad (u,v) \in \textit{edges}(a) \\ \uparrow & \text{otherwise} \end{cases} \end{aligned} \tag{3.59}$$

Finally, function $\textit{trans-sockets} : \mathcal{C}_{\mathcal{M}} \times \mathcal{U} \nrightarrow \wp(\mathcal{U})$ in Eq. 3.60 returns the socket places of a particular substitution transition.

$$\begin{aligned} &\forall a \in \mathcal{C}_{\mathcal{M}} \,.\, \forall t \in \mathcal{U} \,.\, \forall u \in \mathcal{U} \\ &\quad u \in \textit{trans-sockets}(a,t) \triangleq\!\!\Longleftrightarrow \\ &\qquad a = (C, T_{sub}, P_{port}, PT) \ \wedge \ \textit{is-substitution}(a,t) \wedge \\ &\qquad \textit{is-port}(a,u) \ \wedge \ ((t,u) \in \textit{edges}(a) \ \vee \ (u,t) \in \textit{edges}(a)) \end{aligned} \tag{3.60}$$

Given the definition of CPNs module, the definition of hierarchical CPNs can be given as follows.

Definition 3.37 (Hierarchical CPNs). A *hierarchical coloured Petri net* is a tuple $\langle S, SM, PS, FS \rangle$ that belongs to the set $\mathcal{H}$ defined as follows:

$$\begin{aligned} &\forall a \in \mathcal{U} \,.\, a \in \mathcal{H} \triangleq\!\!\Longleftrightarrow \\ &\quad a = \langle S, SM, PS, FS \rangle \wedge \\ &\quad S \subseteq \mathcal{C}_{\mathcal{M}} \quad \wedge \quad SM : T_{sub} \to S \quad \wedge \\ &\quad PS : T_{sub} \times P_{sock} \to S \times P_{port} \quad \wedge \quad FS :\subseteq 2^{P} \ \} \end{aligned} \tag{3.61}$$

such that:

- $S \subseteq \mathcal{C}_{\mathcal{M}}$ is a finite set of CPNs module. It is required that $\forall x, y \in S \,.\, x \neq y \,.\, (\textit{places}(x) \cup \textit{transitions}(x)) \cap (\textit{places}(y) \cup \textit{transitions}(y)) = \varnothing$.
- $SM : T_{sub} \to S$ is a submodule function that assigns a submodule to each substitution transition. It is required that the module hierarchy is acyclic.
- $PS : T_{sub} \times P_{sock} \to S \times P_{port}$ is a port-socket relation function that assigns to each socket of a substitution transition t a port of its corresponding submodule $SM(t) \in \mathcal{C}_{\mathcal{M}}$. It is required that $\forall s \in S \,.\, \forall t \in T_{sub} \,.\, \forall (p,q) \in P_{sock}(t) \times P_{port}(SM(t)) \,.\, \textit{socket-type}(s,t,p) = \textit{port-type}(s,q) \wedge \textit{colour-set}(s,p) = \textit{colour-set}(s,q) \wedge \textit{init}(s,p) = \textit{init}(s,q)$.
- $FS \subseteq 2^{P}$ is a set of non empty fusion sets such that $\forall f \in FS \,.\, \forall p, q \in f \,.\, \textit{colour-set}(p) = \textit{colour-set}(q) \wedge \textit{init}(p) = \textit{init}(q)$.

The basic access functions are summarized in table Tab. 3.7 and they are defined as follows: for any hierarchical CPNs module $a \in \mathcal{H}$ such that $a = (S, SM, PS, FS)$, $\textit{modules}(a) = S$, and $\textit{compound-places}(a) = FS$.

The function $\textit{submodule} : \mathcal{H} \times \mathcal{U} \to \mathcal{U}$ returns the submodule associated to a particular substitution transition. Its behaviour is defined in Eq. 3.62.

Table 3.7. Basic operations on hierarchical CPNs.

Symbol	Function	Description	Ref
	$modules : \mathcal{H} \to \wp(\mathcal{C}_{\mathcal{M}})$	The contained CPNs modules.	*inline*
	$compound\text{-}places : \mathcal{H} \times \to \wp(\mathcal{U})$	The set of compound places (fusion sets) in the net.	*inline*
	$submodule : \mathcal{H} \times \mathcal{U} \to \mathcal{U}$	The submodule associated to a particular substitution transition.	3.62
	$module\text{-}port : \mathcal{H} \times \mathcal{U} \times \mathcal{U} \times \mathcal{C}_{\mathcal{M}} \to \mathcal{U}$	The module port associated to a substitution transition socket.	3.63

$$
\begin{aligned}
&submodule : \mathcal{H} \times \mathcal{U} \to \mathcal{U} \ \ is \\
&\quad \forall a \in \mathcal{H} \,.\, \forall u \in \mathcal{U} \,. \\
&\qquad submodule(a,u) = \begin{cases} m & \text{if } m \in S \wedge \\ & \quad \exists t \in \bigcup_{s \in S} sub\text{-}transitions(s) \,.\, SM(t) = m \\ \uparrow & \text{otherwise} \end{cases}
\end{aligned} \tag{3.62}
$$

The function $module\text{-}port : \mathcal{H} \times \mathcal{U} \times \mathcal{U} \times \mathcal{C}_{\mathcal{M}} \to \mathcal{U}$ returns the module port associated to a particular substitution transition socket. It is defined in Eq. 3.63.

$$
\begin{aligned}
&module\text{-}port : \mathcal{H} \times \mathcal{U} \times \mathcal{U} \times \mathcal{C}_{\mathcal{M}} \to \mathcal{U} \ \ is \\
&\quad \forall a \in \mathcal{H} \,.\, \forall t, u \in \mathcal{U} \,.\, \forall m \in \mathcal{C}_{\mathcal{M}} \,. \\
&\qquad module\text{-}port(a,t,u,m) = \begin{cases} p & \text{if } t \in \bigcup_{s \in S} sub\text{-}transitions(s) \,.\, SM(t) = m \wedge \\ & \quad v \in trans\text{-}sockets(a,t) \wedge PS(t,v,u) = p \\ \uparrow & \text{otherwise} \end{cases}
\end{aligned} \tag{3.63}
$$

Markings

One of the main differences between low level Petri nets and CPNs is the notion of token that in the latter case has its own identity because it carries a data value. In the following, the notion of marking, binding, and binding element are initially stated for a non-hierarchical CPNs model, then such definitions are extended to hierarchical CPNs modules.

Definition 3.38 (Marking). Given a CPNs model $a \in \mathcal{C}$, a marking is a function $q : \mathcal{U} \to \mathcal{M}$ that maps each place $p \in places(a)$ to a multiset of values $q(p) \subseteq colour\text{-}sets(p) \subseteq \mathcal{M}$ representing the state of p. The individual elements in the multiset are called tokens. The set of all markings is denoted by $\mathcal{Q}$.

The notion of marking can be extended to the case of hierarchical CPNs models by considering the function *compound-places* instead of *places* in the previous definition.

From the notion of marking it is clear that the values of the tokens contained in a place p are required to match the type of the place.

Definition 3.39 (Binding). A *binding* $b : \mathcal{U} \to \mathcal{U}$ of a transition t is a function that maps each variable $v \in \mathit{variables}(t)$ to a value $b(v) \in \mathit{colour\text{-}set}(v)$ belonging to the type of the variable v. The set of all bindings for a transition t is denoted as $\mathcal{B}(t)$, while the set of all binding elements in a CPNs model is denoted as $\mathcal{B}$.

Definition 3.40 (Binding Element). A *binding element* is a pair $(t, b) \in \mathcal{U} \times \mathcal{B}$ consisting of a transition t and a binding b of t. The set of all binding elements for a transition t is denoted as $\mathcal{B}e(t)$, while the set of binding elements in a CPN model is denoted as $\mathcal{B}e$.

The concepts of binding and binding element can be extended to a hierarchical CPNs model by substituting in the previous definitions the notion of transition with the notion of transition instance for all transitions that are not substitution transitions.

Definition 3.41 (Step). A *step* is a finite non-empty multiset of binding elements. In the following the set of all steps is denoted as $\mathcal{M}_{be}$.

Given the notions of binding element and step, we can now define when a biding element and a step are enabled in a given marking. For this purpose, for any expression $e \in \mathcal{E}$, we write $e\langle b \rangle$ to denote the evaluation of the expression e considering the binding b.

Definition 3.42 (Enabled Binding Element). A binding element $(t, b) \in \mathcal{B}e$ is enabled in a marking $q \in \mathcal{Q}$ if and only if the transition guard $\mathit{guard}(t)$ associated to t is satisfied when evaluated in the binding b, and there are sufficient tokens in the input places of the transition. The last condition requires that for each place p connected to t, the multiset of tokens obtained by evaluating the arc expression $\mathit{arc\text{-}expr}(p, t)$ in the binding b is contained in the multiset of tokens $q(p)$ representing the current marking of p. This concept is captured by the function $\mathit{is\text{-}enabled} : \mathcal{C} \times \mathcal{B}e \times \mathcal{Q} \to \mathbb{B}$ defined as:

$$
\begin{aligned}
&\forall a \in \mathcal{C} \,.\, \forall (t, b) \in \mathcal{B}e \,.\, \forall q \in \mathcal{Q} \,. \\
&\quad \mathit{is\text{-}enabled}(a, (t, b), q) \stackrel{\Delta}{\Longleftrightarrow} \\
&\qquad \mathit{guard}(t)\langle b \rangle = \mathtt{T} \wedge \\
&\qquad \forall p \in \mathit{places}(a) \,.\, \mathit{arc\text{-}expr}(p, t)\langle b \rangle \subseteq q(p)
\end{aligned}
$$

The definition can be extended for a hierarchical CPNs model by considering for the first condition the guard of the all instances of the transitions that are not substitution transitions, and for the second condition the union of the tokens contained in each compound place. When a binding element (t, b) is enabled in a marking $q \in \mathcal{M}$, it may *occur* leading to a marking r defined as in the following definition.

Definition 3.43 (Next Marking). The execution of a binding element $(t, b) \in \mathcal{B}e$ starting from a marking $q \in \mathcal{Q}$ produces a new marking $r \in \mathcal{Q}$ such that:

$$
\forall p \in \mathit{places}(a) \,.\, r(p) = q(p) \setminus \mathit{arc\text{-}expr}(p, t)\langle b \rangle \cup \mathit{arc\text{-}expr}(t, p)\langle b \rangle \tag{3.64}
$$

Definition 3.44 (Enabled Step). A step $M \in \mathcal{M}_{be}$ is enabled in a marking $q \in \mathcal{Q}$ if and only if each binding element composing the step is enabled. In other words, if the guard associated to each transition in the step is satisfied, and there are sufficient tokens in the input places of each involved transition. These conditions are captured by the following function.

$$\begin{aligned} &\forall a \in \mathcal{C} \,.\, \forall M \in \mathcal{M}_{be} \,.\, \forall q \in \mathcal{Q} \,. \\ &\quad \mathit{is\text{-}enabled\text{-}step}(a, M, q) \stackrel{\Delta}{\Longleftrightarrow} \\ &\qquad \forall (t, b) \in M \,.\, \mathit{is\text{-}enabled}(a, (t, b), q) \end{aligned} \tag{3.65}$$

The notion of enabled step can be extended to a hierarchical CPNs model by considering compound places and transition instances, instead of places and transitions.

Definition 3.45 (Firing Rule). The execution of a step $M \in \mathcal{M}_{be}$ starting from a marking $q \in \mathcal{Q}$ produces a new marking $r \in \mathcal{Q}$ defined as follows:

$$\forall p \in \mathit{places}(a) \,.\, r(p) = q(p) \setminus \sum_{(t,b) \in M} \mathit{arc\text{-}expr}(p, t)\langle b \rangle \cup \sum_{(t,b) \in M} \mathit{arc\text{-}expr}(t, p)\langle b \rangle \tag{3.66}$$

◇

3.4 Yet Another Workflow Language

> "*The original formal semantics of YAWL were specified as a transition system, but more recently, the CPN formalism was used to provide an operational semantics for newYAWL.*"
> – Surmounting BPM challenges: the YAWL story [67]

Workflow Management Systems (WfMSs) or more recently Process-Aware Information Systems (PAISs) are software systems driven by an explicit workflow or process specification. Roughly speaking a PAIS provides a graphical process modeling language (PML), an editor, and a so called engine.

The *graphical PML* is tailored for describing business processes (BPs) in terms of simple tasks constrained by a particular ordering relation. The *editor* is used to design new process specifications or adjust existing ones to new contingent needs. The *engine* is the run-time environment in which process specifications are executed. An engine can manage several process instances simultaneously, each one related to a particular process specification version. The engine has to balance existing resources and coordinating the available human agents in performing their activities. The interaction with the end-user occurs through custom or automatically generated graphical user interfaces.

The Yet Another Workflow Language (YAWL) [15, 68] together with its reference implementation YAWL System [69] are a good example of such architecture. YAWL is a graphical PML tailored for workflow and business process design. This language was born as an academic research project in the context of the Workflow Patterns Initiative [28] to show how a real software system can provide a comprehensive support for workflow patterns [67].

The original YAWL language [68] may be considered an evolution of workflow nets and reset workflow nets [67] where the basic Petri nets elements are enriched with new control-flow constructs, among which we can cite: or-split and or-join for advanced synchronization patterns, multiple instances, cancellation regions, and more compact syntax to express routing patterns [67].

YAWL System [69] is a full-fledged WfMS built around the YAWL language and can be actually considered its reference implementation. In particular, the original YAWL language has been enriched with all those features needed to make it executable, e.g. a way to declare and transform data, as well as a mechanism to pass them around task, or the definition of scoping rules [15].

Following the evolution of workflow patterns [29], a revised version of YAWL, called newYAWL, came to life [70, 71]. This new language version covers missing patterns with additional control-flow constructs, like thread split and merge; however, for now such constructs are not yet supported by the YAWL reference implementation.

3.4.1 Graphical Elements and Syntax

The graphical elements of YAWL language with their inscriptions are summarized in Fig. 3.18, where component names are highlighted in bold. Neither the size of the graphical elements, nor the position of the inscriptions is relevant, provided that

all inscriptions are close to their related constructs. Models are usually oriented from left to right, hence by convention labels are placed below the components.

A start element is depicted as a circle containing a triangle while a stop element is depicted as a circle containing a square as in Fig. 3.18.a and Fig. 3.18.b, respectively. A choice is represented as an empty circle like in Fig. 3.18.c.

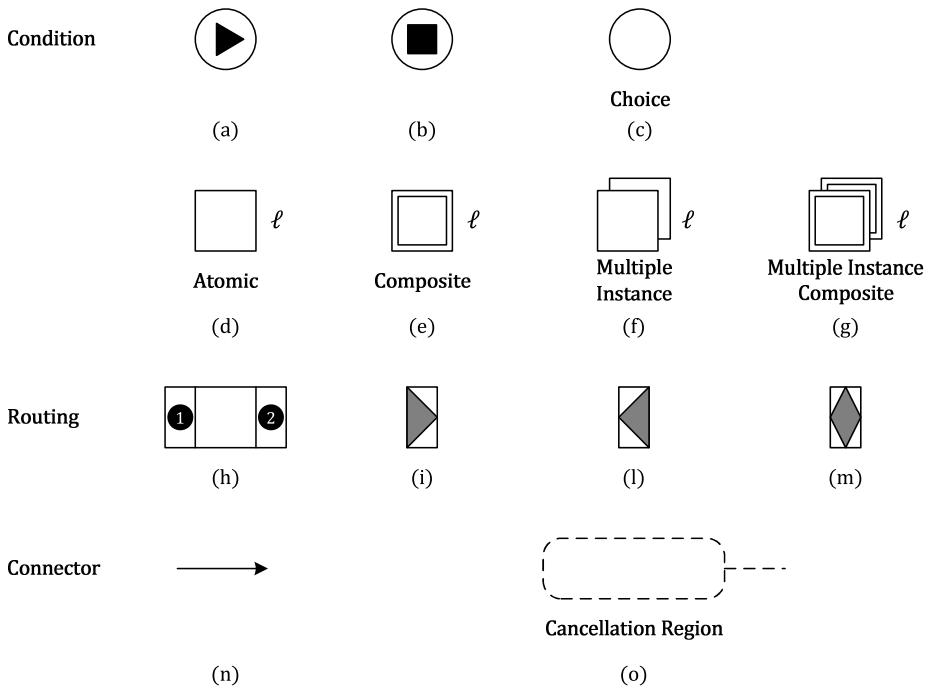


Fig. 3.18. YAWL language elements.

Tasks have a rectangular form enriched with an optional label ℓ placed near to it, as in Fig. 3.18.d. In the YAWL implementation [69] the label can represent a generic task name or a unique identifier: in the former case the identifier is implicit. Similarly, a composite task, representing the invocation of a sub-net defined elsewhere, is depicted as a square with a double line border, as in Fig. 3.18.e. A multiple instance task is represented using two partially overlapping squares, as in Fig. 3.18.f, while a multiple instance composite task is represented by two partially overlapping multiple instance symbols, as shown in Fig. 3.18.g.

Tasks are connected through arcs representing control-flow relations, depicted using straight or curved lines with a solid arrow at one end, such as the one in Fig. 3.18.n. The flow of control can also be modified by enriching tasks with some routing elements. In particular, a routing element can be attached at the beginning and/or at the end of a task, as illustrated in Fig. 3.18.h, where labels 1 and 2 in white on black point out where routing elements can be placed. Such labels are not part of the language and the two positions have to be inferred from the connected arcs: the first position is on the side of the incoming arc, while the second position

is on the side of the outgoing arc. The available routing elements are those from Fig. 3.18.i to Fig. 3.18.m. If the element in Fig. 3.18.i is placed in position 1, the task is preceded by an *and-join* construct, while if it is placed in position 2, the task is followed by a *xor-split*. If the element in Fig. 3.18.l is placed in position 1, the task is preceded by a *xor-join*, while if it is placed in position 2, the task is followed by an *and-split* task. Finally, if the element in Fig. 3.18.m is placed in position 1, the task is preceded by an *or-join* task, while if it is placed in position 2, the task is followed by an *or-split* task.

The last graphical element is the *cancellation region* in Fig. 3.18.o, it is composed of a rounded box with dashed line border which can contain any kind of graphical elements, and a dashed line arc that connect the rounded box to the triggering task. In the reference implementation cancellation regions are not explicitly shown in the model.

The YAWL syntax requires that each net is a graph containing a finite set of tasks connected by arcs, such that each task is in a path from the start symbol to the end symbol. Each task can have only one incoming arc and one outgoing arc, unless some routing elements have been attached to it. Relatively to Fig. 3.18.h, if a routing element has been attached in position 1, then multiple incoming arcs can be connected to the task, while if a routing element has been attached in position 2, then multiple outgoing arcs can be connected to it. A condition can be placed between any two tasks and it can have multiple incoming and outgoing arcs. Each cancellation region shall be connected to a transition and can contain any graphical element, except for the connected transition, the start and the end elements.

YAWL system allows one to define some data aspects associated to a model through a set of net variables. Net variables are visible to all tasks of a net. Tasks can read and write such shared variables providing a primitive communication mechanism that will be further described in the remainder of this section. In YAWL no graphical notation is given for net variables, as well as for all the other language constructs related to data manipulation. However, in order to give a complete graphical representation of an executable model, in the following some additional notations shall be introduced.

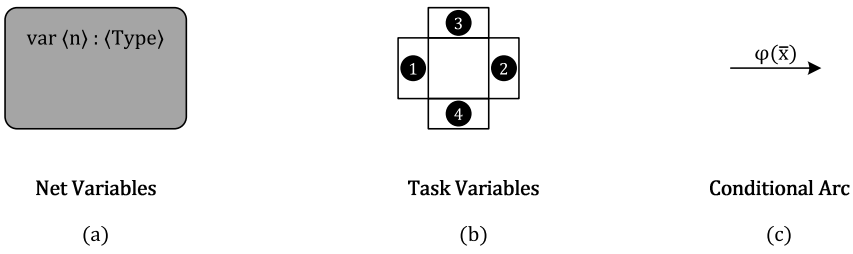


Fig. 3.19. Graphical representation of the YAWL data aspects.

Net variables are declared in a list near to the net, as in Fig. 3.19.a, in which each declaration takes the form `var` $\langle n \rangle : \langle \mathit{Type} \rangle$, where `var` is a keyword, n is a unique valid identifier, and *Type* identifies the range of acceptable values. The

mapping between global and task variables are specified as in Fig. 3.19.b: the input mappings are specified in the box containing the white on black label 3, if the model orientation is from left to right, or in the box containing the white on black label 1 if the orientation is from top to bottom. Similarly, output mappings are specified in the box containing the white on black label 4, if the model orientation is from left to right, or in the box containing the white on black label 2 if the orientation is from top to bottom. Each mapping has the form $x \leftarrow y$, if it is an input mapping x is a local task variable and y is a global net variable, conversely if it is an output mapping x is a net variable and y is a local task variable.

Depending on the involved routing constructs, incoming and outgoing arcs can have an associated boolean condition. Similarly to net variables, conditions are not explicitly represented in the YAWL reference implementation. Conversely, here an arc is annotated as in Fig. 3.19.c, where $\varphi(\bar{x})$ is a boolean formula defined on a set $\bar{x}$ of existing net variables.

3.4.2 Language Interpretation

A YAWL net is similar to a Petri net in which places can been omitted and tasks can be directly connected through arcs. However, the YAWL language contains some specific constructs whose behavior cannot be easily replicated by a low-level Petri net, such as the *or-join*, net variables, and conditions that are managed and evaluated at run-time. Finally, even if tasks can be compared to Petri nets transitions, an important difference exists between them: transitions are assumed to be instantaneous, while tasks represent real activities with a particular duration that clearly requires time.

YAWL has a token-based semantics routed on Petri nets, even if such tokens are not explicitly represented. Tokens are threads of control that flow inside the net enabling the encountered tasks. In general, a task is *enabled* when it receives the necessary tokens required by the routing constructs attached to it. Excluding routing constructs, an *atomic* or *composite task* is enabled when a token is present in its incoming arc. An enabled atomic or composite task can be activated and start its execution by suspending the current thread of control and releasing it at completion by putting a new token on its outgoing arc. An atomic task can be seen as a function $\bar{x} \leftarrow f(\bar{y})$ that reads a set of net variables $\bar{y}$ in input, performs some operations on them eventually using other local variables, and then stores the final outputs into a set of net variables $\bar{x}$, not necessarily different from $\bar{y}$. Similarly, a composite task can be seen as a function composition.

A *multiple instance task* denotes the parallel executions of the same atomic task multiple times: some or all these instances should need to be synchronized before the thread of control can pass to the subsequent tasks. Finally, a *multiple instance composite task* represents the parallel execution of several instances of the same composite task defined elsewhere. A multiple instance (composite) task is enabled when a token is present in its incoming arc; an enabled multiple instance (composite) task generates a token for each instance that has to be executed: then the generated tokens are synchronized at completion and a unique token is placed in the outgoing arc.

Special treatment should be reserved to tasks with routing constructs which have different activation behaviour. An *and-split* is enabled when its related task

completes and it generates a token on each outgoing arc, producing multiple concurrent threads of control. A *xor-split* is enabled in the same way, but it produces a token in only one of its outgoing arcs, on the basis of the evaluation of the condition associated to each of them; if no condition is true, the last arc is considered the default one and a token is produced in it anyway. Finally, an *or-split* is enabled as the other split constructs, but it produces a token on each outgoing arcs for which the corresponding condition evaluates to true; at least one condition has to be true, otherwise the default arc is enabled; conversely not all arc conditions have to be true, since the default arc is disabled if at least one condition is true.

An *and-join* is enabled when a token is present in each incoming arc, while a *xor-join* is enabled when a token is present in any of its incoming arcs; finally, an *or-join* is enabled when at least one of its incoming arcs contains a token and it is not possible to reach a state where an not yet marked arc also receives a token and all incoming arcs that were already marked remain marked. When a join construct is enabled, the attached task is also enabled.

A *choice* can be used between any two or more tasks and it assumes different meanings on the basis of the specific configuration. If a choice has a single incoming and a single outgoing arc, it has the same function as a place between two transitions and it can be safely removed. If a choice has several incoming arcs and only one outgoing arc, it works as an *xor-join*, because it is enabled as soon as a thread of control is available in any of its incoming arcs. Finally, a choice with several outgoing arcs is used to represent the behavior of a deferred choice. In this case the choice is enabled when a token is present in its incoming arc, but the choice of on which outgoing arc the token will be placed is not taken by the system, but is postponed and leaved to the user. In other words, while with a *xor-split* only one of the connected tasks will become enabled, with a deferred choice all the connected tasks are enabled until one of them is chosen, after this choice the other tasks are withdrawn.

The *cancellation region* can contain any element of the net, it allows one to define the cancellation of individual tasks, arcs, portions of a process or even an entire process. It is always attached to a specific task, when this task completes any thread of control residing inside the cancellation region is aborted and any executing task within the cancellation region is terminated.

A net terminates when the *end* element is reached by a token. As soon as this happens all the remaining threads of control and running tasks are terminated.

3.4.3 Language Formalization

YAWL was originally born as a compact notation for specifying workflows [72]. Its semantics has been given by mapping its constructs on classical Petri nets, then transition systems and CPNs models as long as more complex constructs was added [67]. Subsequently, YAWL was evolved into a full-fledged PML and workflow system, by integrating all necessary constructs for data manipulation, resource management, user interface generation, and so on.

Although YAWL semantics is explained in terms of Petri nets, its formal semantics is officially given through (labeled) transition systems [68]. The reference implementation can help in figure out how YAWL works when the formalization does not contains the needed information.

In 2009 newYAWL has been defined by adding new constructs for covering the remaining workflow patterns. The semantics of newYAWL and its run-time support is claimed to be given in terms of CPNs [71]. Anyway, the complete CPNs model has not been officially published, as pointed out by Börger in [36], probably due to its complexity. For this reasons, it is not possible to give here the complete formalization of the YAWL language. Instead, this section focuses on control-flow aspects with special attention to the *or-join* semantics, whose formalization is particularly challenging.

In the following the set of YAWL component types is denoted as $\mathcal{Y}_{types}\{\mathsf{AT}, \mathsf{CT}, \mathsf{MI}, \mathsf{MIC}, \mathsf{C}, \mathsf{AS}, \mathsf{AJ}, \mathsf{OS}, \mathsf{OJ}, \mathsf{XS}, \mathsf{XJ}, \mathsf{SA}, \mathsf{SO}\}$, where the symbols are defined in Tab. 3.8.

Definition 3.46 (YAWL net). A YAWL net is a tuple (V, E, Y, A, L, R) that belongs to the set $\mathcal{Y}$ defined as follows:

$$\begin{aligned} \forall a \in \mathcal{U} \,.\, a \in \mathcal{Y} \triangleq\Longleftrightarrow \\ a = (V, E, Y, A, L, R) \wedge \\ (V, E) \in \mathcal{G} \quad \wedge \quad Y : V \to \mathcal{Y}_{types} \quad \wedge \\ L : T \to A \cup \{\tau\} \quad \wedge \quad R : T \to \wp(V \times E) \end{aligned} \tag{3.67}$$

such that:

- $(V, E) \in \mathcal{G}$ is a finite directed graph such that each vertex must be in a path from the start to the end element. Formally: $\exists s \in V \,.\, Y(s) = \mathsf{SA} \wedge \exists z \in V \,.\, Y(z) = \mathsf{SO} \wedge \forall z \in V \,.\, \exists \{v_i\}_{i=1}^{n} \subseteq V \,.\, v_1 = s \wedge v_n = z \wedge \forall i \in [1, n-1] \,.\, (v_i, v_{i+1}) \in E$.
- $Y : V \to \mathcal{Y}_{types}$ is a total function that associates to each vertex its type. The function Y induces a partition on the set V, in particular the set $T = v \in V \mid Y(v) \in \{\mathsf{AT}, \mathsf{CT}, \mathsf{MI}, \mathsf{MIC}\}$ denotes the set of tasks.
- A is a set of labels such that $\tau \notin A$, where τ denotes that the label is missing.
- $L : T \to A \cup \{\tau\}$ is a total function that associates to each task its name.
- $R : T \to \wp(V \cup E)$ is the function that associate to each task its cancellation region that may include tasks and arcs.

Table 3.8. YAWL component types in $\mathcal{Y}_{types}$.

Symbol	Meaning	Symbol	Meaning
AT	atomic task	**CT**	composite task
MI	multiple instance task	**MIC**	multiple instance composite task
C	condition	**AS**	and split
AJ	and join	**OS**	or split
OJ	or join	**XS**	xor split
XJ	xor join	**SA**	start
SO	stop		

The basic access functions are summarized in Tab. 3.9 and they are defined as follows: for any $a \in \mathcal{Y}$ such that $a = (V, E, Y, A, L, R)$, $vertices(a) = V$, $edges(a) = E$, Function $type : \mathcal{Y} \times \mathcal{U} \nrightarrow \mathcal{Y}_{types}$ associates to each vertex in the net its type, its

Table 3.9. Basic operations on YAWL nets.

Symbol	Function	Description	Ref
	$vertices : \mathcal{Y} \to \wp(\mathcal{U})$	Vertices of the underlying graph.	*inline*
	$edges : \mathcal{Y} \to \wp(\mathcal{U})$	Edges of the underlying graph.	*inline*
	$type : \mathcal{Y} \times \mathcal{U} \nrightarrow \mathcal{Y}_{types}$	Vertex type.	Eq. 3.68
	$tasks : \mathcal{Y} \to \wp(\mathcal{U})$	Set of task elements.	Eq. 3.69
	$label : \mathcal{Y} \times \mathcal{U} \nrightarrow A \cup \{\tau\}$	Task label.	Eq. 3.70
	$canc\text{-}region : \mathcal{Y} \times \mathcal{U} \nrightarrow \wp(\mathcal{U})$	Cancellation region associated to the given transition.	Eq. 3.71
	$components : \mathcal{Y} \times \mathcal{U} \to \wp(\mathcal{U})$	Set of components reachable from the start element.	Eq. 3.72

behaviour is reported in Eq. 3.68.

$$\begin{aligned} &type : \mathcal{Y} \times \mathcal{U} \nrightarrow \mathcal{Y}_{types} \ \textit{is} \\ &\quad \forall a \in \mathcal{Y} \,.\, u \in \mathcal{U} \,.\, a = (V, E, Y, A, L, R) \,. \\ &\qquad type(a, u) = \begin{cases} Y(u) & \text{if } u \in vertices(a) \\ \uparrow & \text{otherwise} \end{cases} \end{aligned} \tag{3.68}$$

Function $tasks : \mathcal{Y} \to \wp(\mathcal{U})$ returns the set of YAWL components that are atomic, composite, multiple instance, or multiple instance composite tasks. Its behavior is formally defined in Eq. 3.69

$$\begin{aligned} &tasks : \mathcal{Y} \to \wp(\mathcal{U}) \ \textit{is} \\ &\quad \forall a \in \mathcal{Y} \,.\, u \in \mathcal{U} \,.\, u \in tasks(a) \overset{\triangle}{\Longleftrightarrow} \\ &\qquad a = (V, E, Y, A, L, R) \wedge \\ &\qquad type(a, u) \in \{\mathbf{AT}, \mathbf{CT}, \mathbf{MI}, \mathbf{MIC}\} \end{aligned} \tag{3.69}$$

Function $label : \mathcal{Y} \times \mathcal{U} \nrightarrow A \cup \{\tau\}$ returns the label associated to each task in the net. Its behavior is defined in Eq. 3.70.

$$\begin{aligned} &label : \mathcal{Y} \times \mathcal{U} \nrightarrow A \cup \{\tau\} \ \textit{is} \\ &\quad \forall a \in \mathcal{Y} \,.\, u \in \mathcal{U} \,.\, a = (V, E, Y, A, L, R) \,. \\ &\qquad label(a, u) = \begin{cases} L(u) & \text{if } u \in tasks(a) \\ \uparrow & \text{otherwise} \end{cases} \end{aligned} \tag{3.70}$$

Function $canc\text{-}region : \mathcal{Y} \times \mathcal{U} \nrightarrow \wp(\mathcal{U})$ is the function that associates to each transition its cancellation region. Its behavior is defined in Eq. 3.71.

$$\begin{aligned} &canc\text{-}region : \mathcal{Y} \times \mathcal{U} \nrightarrow \wp(\mathcal{U}) \ \textit{is} \\ &\quad \forall a \in \mathcal{Y} \,.\, u \in \mathcal{U} \,.\, a = (V, E, Y, A, L, R) \,. \\ &\qquad canc\text{-}region(a, u) = \begin{cases} R(u) & \text{if } u \in tasks(a) \\ \uparrow & \text{otherwise} \end{cases} \end{aligned} \tag{3.71}$$

Finally, function $components : \mathcal{Y} \times \mathcal{U} \to \wp(\mathcal{U})$ returns the components of a YAWL net that are reachable from the start element, as formally described in Eq. 3.72.

$$
\begin{aligned}
&components : \mathcal{Y} \times \mathcal{U} \to \wp(\mathcal{U}) \ \ is \\
&\quad \forall a \in \mathcal{Y} \,.\, \forall u \in vertices(a) \,.\, type(a,u) = \textsf{SA} \,. \\
&\quad\quad \forall v \in \mathcal{U} \,.\, v \in components(a,u) \stackrel{\triangle}{\Longleftrightarrow} \\
&\quad\quad\quad u = v \vee \\
&\quad\quad\quad \exists x \in components(a,u) \,.\, (x,v) \in edges(a)
\end{aligned}
\tag{3.72}
$$

$\diamond$

3.5 Business Process Model and Notation

> "*The execution semantics are described informally (textually), and this is based on prior research involving the formalization of execution semantics using mathematical formalisms.*"
> – BPMN 2.0 Specification [16]

The Business Process Modeling Notation (BPMN), recently renewed as Business Process Model and Notation [16], is an Object Management Group (OMG) standard that provides among all the following features: (1) an object model suitable for the business process domain; (2) a wide range of graphical constructs for defining business process diagrams; (3) an XML-based interchange format for enhancing vendor tools compatibility; (4) an executable semantics for BP diagrams; (5) a way for translating BP models into the Web Services Business Process Execution Language (WS-BPEL).

The BPMN standard treats so many aspects about business process modeling, that there is no way to produce an introduction that is both compact and exhaustive. To give an idea about this, consider that the latest BPMN specification [16] is about five hundred pages long, and one of its parts is directly related to the WS-BPEL specification [73], which alone contains over than two hundred pages, without considering auxiliary specifications. For such reason, this section focuses only on those aspects of the BPMN standard related to the design of internal processes inside a single organization, while other aspects related to the coordination of multiple processes, like choreographes and orchestrations, are not treated.

3.5.1 Graphical Elements and Syntax

The main graphical elements of BPMN core language are summarized in Fig. 3.20, and collected into five groups: event, task, gateway, data, and connector. These groups are part of the more general syntactical categories defined in [16]: flow objects, data, connecting objects, swim lanes, and artifacts.

Flow objects includes the main graphical elements for defining the behaviour of a process, such as: events, tasks, and gateways. Several types of events can be found in a BPMN model, their representation changes on the basis of the position in which they are placed and the nature of the event. A *start event* is represented like in Fig. 3.20.a, as a circle with a thin border and an empty content, while a generic *intermediate event* is characterized by a double line border as in Fig. 3.20.b, and an *end event* by a thicker border as in Fig. 3.20.c. A particular variant of end event is the *stop event* which is represented, as in Fig. 3.20.g, by placing a filled circle inside an end event element. Other kinds of events can be represented by placing a specific icon inside these circles; for instance, Fig. 3.20.e depicts a *message event*, Fig. 3.20.f depicts a *timer event*, and finally Fig. 3.20.g depicts a *error event*.

An *atomic* task is depicted as a rounded rectangle as exemplified in Fig. 3.20.h, a *sub-process* or composite task is depicted in a similar way, but it is characterized by a small square with an inner plus symbol placed on the bottom of the rounded rectangle, as in Fig. 3.20.i.

An *exclusive gateway* is depicted as a diamond containing a cross, an *inclusive gateway* is characterized by a circle inside the diamond, a *parallel gateway* contains a plus symbol inside the diamond, while the *decision gateway* has a double

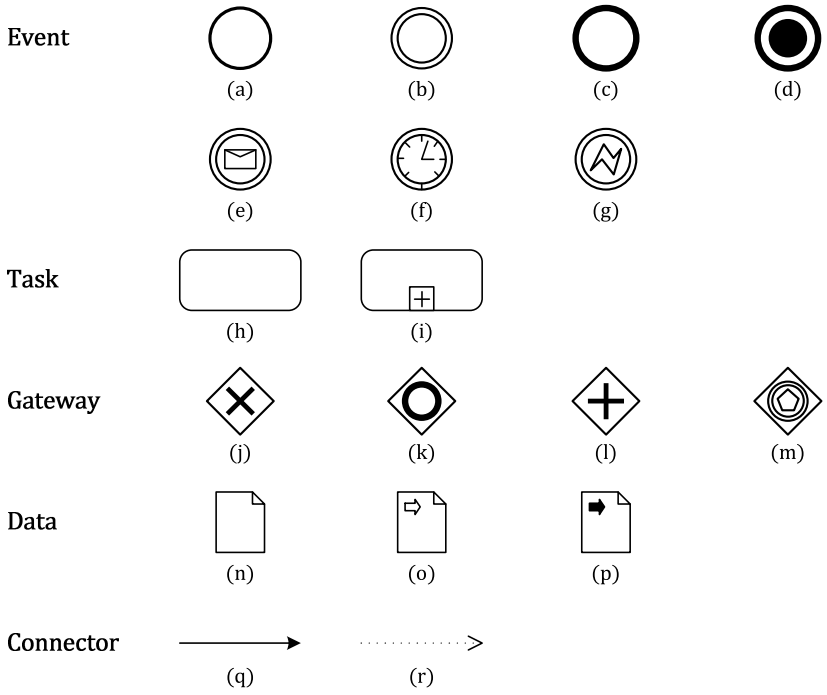


Fig. 3.20. BPMN main graphical elements.

circle with an empty pentagon inside the diamond, as shown from Fig. 3.20.l to Fig. 3.20.m, respectively.

A *data object* produced or consumed by a task is represented as a paper sheet with a folded upper-right corner, like the one in Fig. 3.20.n. Data objects consumed or produced by a process or task are represented by placing an empty or filled arrow inside the paper, respectively, as happens in Fig. 3.20.o and Fig. 3.20.p.

Two kinds of *connecting objects* are considered here: sequence flow and association. A *sequence flow* is depicted with a solid line as in Fig. 3.20.q, while an *association* is depicted as a dotted line with an open arrowhead, as in Fig. 3.20.r, which connects a data object to a task and vice-versa.

The BPMN elements for message exchange are not considered here, because they cannot be used to model a single process, but only to represent the interaction among several processes, as stated in [16]: "A message flow *must* connect two separate pools. They connect either to the pool boundary or to flow objects within the pool boundary. They *must not* connect two objects within the same pool". A pool is a graphical representation of a participant; hence, no message exchanges can be defined inside the same process. The only considered message-related construct is the message event, in order to represent the case in which an event is waited from the external environment, but no further assumptions are given about the event generation. Graphical elements related to role management are not considered either, because of marginal interest for the core BPMN semantics. As a consequence, elements such as pools and swim lanes are omitted. Artifacts are also discarded in

this introduction, since they are used only for documentation purposes, and they do not influence the execution.

With such assumptions at hand, a BPMN net is a directed graph containing a finite number of tasks, gateways and events connected through sequence flows. Moreover, tasks can be associated through an association arc to a data object element. Even if the BPMN specification [16] does not preclude the ability of connecting a task or event with several incoming or outgoing sequence flows, here it is assumed that at most one flow can enter a task or event, and only one flow can exit from it. This is reasonable, because the same logic induced by multiple sequence flows can be implemented with one or more gateways. Furthermore, each net shall start with one or more start events; a start event cannot have an incoming flow but only one outgoing flow, and shall end with one or more end or termination events, which can have only an incoming flow and no outgoing flows. A gateway can have several incoming branches and one outgoing branch, or one incoming branch and several outgoing branches.

3.5.2 Language Interpretation

Similarly to the previous languages, the BPMN informal semantics can be explained in terms of tokens that flows inside the net activating tasks. A token is nothing more than a thread of control, it is ideally produced by each start event and injected into the net through the outgoing sequence flow. Note that two start events are alternative: a process instance is triggered by one of the start events and it does not wait for the other ones. An atomic or composite task is *ready* when the required number of tokens is available in its incoming sequence flow. A task changes its status from ready to *active*, when the necessary data inputs become available. In other words, a ready task will be executed only when its input data becomes available. Finally, when all associated operations have been performed, the task changes its status in *complete* and the specified number of tokens is produced through its outgoing sequence flow, generating the required data outputs.

As regards to data aspects, data objects are the primary construct for modeling data within a process or task. The lifecycle of a data object is strictly related to the process or task lifecycle: when a process or task is instantiated, all data object instances contained within it are also instantiated; similarly, when a process or task is disposed, the data object instances contained inside it are also disposed. Moreover, each process or task can define one or more *input-set* and *output-set*, representing its interface. Each *input-set* and *output-set* refers to zero or more data input and data output objects, respectively. The specification [16] requires that a process or task cannot start until the *input-set* is available, while the *output-set* becomes available at task or process completion.

When a thread of control reaches an intermediate event, it is suspended until a particular event occurs: for instance, the reception of a message from the external environment, or a timeout expiration. An end event consumes the incoming tokens, without producing anything. The net execution terminates when one of the following three conditions holds: (1) all tokens have been consumed by the existing end events, or (2) when no other task can be enabled, or (3) immediately when a stop element is reached by a token.

The behaviour of each gateway depends on the number of connected incoming and outgoing flows. An exclusive gateway with one incoming branch and several outgoing branches acts as a split which redirects the received threads of control on only one of its outgoing branches on the basis of the the associated condition evaluation. Assuming an evaluation order from top to bottom or left to right, the first true condition determines the activation of its corresponding branch. At least one default branch has to be specified, otherwise the gateway throws an exception whenever all conditions are false. Conversely, if the exclusive gateway has several incoming branches and only one outgoing branch, it acts a merge which places a token into its outgoing branch as soon as a token is available in any of its incoming branches.

An inclusive *or* gateway with one incoming branch and several outgoing branches represents a conditional split in which more than one outgoing branches can be enabled, on the basis of the evaluation of the associated conditions. As for the exclusive *or* gateway, at least one branch has to be marked as the default one, in order to avoid the raising of an exception. Similarly, if the inclusive *or* gateway has several incoming branches and one outgoing branch, it act as a generic join which waits the completion of all incoming branch that can complete, before producing a token on its outgoing branch.

A parallel gateway with one incoming branch and several outgoing branches represents a fork which generates a new thread of control on its outgoing branches. Conversely, if it has several incoming branches and only one outgoing branch, it denotes a join which waits for the completion of all its incoming branches, before placing a token in its outgoing branch.

Finally, a *decision* gateway with one incoming branch and several outgoing branches represents an inclusive *or* gateway in which the different alternatives have an associated event which determines its activation.

3.5.3 Language Formalization

The formalization of the BPMN semantics is notoriously challenging, probably due to its complexity. The last BPMN specification [16] claims to solve inconsistencies and ambiguities of the previous version, although many questions about the BPMN semantics seem to be left unspecified. For instance, it explicitly states that BPMN is not a data-flow language, despite that the flow of data is represented by means of message exchanges and data artifact associations. Nevertheless, inside the execution semantics, it claims that if no input-set is available for a task, then its execution will wait until this condition is met, and this is a typical data-flow language behaviour.

The latest specification describes the BPMN execution semantics in two different ways: in the first approach the execution semantics is given in textual form with explicit references to the workflow control-flow patterns [29]. Conversely, the second approach is based on a recursive function that translates BPMN elements into WS-BPEL executable code, exploiting the meaning of its constructions.

Börgher in a recent paper [74] discusses several weak points of the standard, claiming the BPMN does not fulfil the intended goals, especially for what concern interoperability, and it fails to be a valid medium for sharing knowledge about the domain among the different stakeholders.

In the remainder of this section a BPMN net is formalized in terms of a graph. For this purpose the set $\mathcal{B}_{types} = \{\mathsf{SE}, \mathsf{IE}, \mathsf{EE}, \mathsf{TE}, \mathsf{AT}, \mathsf{CT}, \mathsf{XG}, \mathsf{OG}, \mathsf{AG}, \mathsf{DO}, \mathsf{DI}, \mathsf{DU}\}$ is introduced which collects all the considered component types defined in Tab. 3.10.

Table 3.10. BPMN component types in $\mathcal{B}_{types}$.

Symbol	Meaning	Symbol	Meaning
SE	start event	IE	intermediate event
EE	end event	TE	stop event
AT	atomic task	CT	composite task
XG	exclusive or gateway	OG	inclusive or gateway
AG	parallel gateway	DG	decision gateway
DO	data object		
DI	data input object	DU	data output object

Definition 3.47 (BPMN net). A BPMN net is a tuple (V, E, Y, X, A, L, D) that belongs to the set $\mathcal{B}$ defined as follows:

$$\begin{aligned} &\forall a \in \mathcal{U} \,.\, a \in \mathcal{B} \triangleq\!\!\Leftrightarrow \\ &\quad a = (V, E, Y, X, A, L, D) \quad \wedge \\ &\quad (V, E) \in \mathcal{G} \quad \wedge \quad Y : V \to \mathcal{B}_{types} \quad \wedge \quad X : E \to \{\mathsf{S}, \mathsf{A}\} \quad \wedge \\ &\quad L : T \to A \cup \{\tau\} \quad \wedge \quad D : T \to \wp(DO \times DI \times DU) \;\} \end{aligned} \tag{3.73}$$

such that:

- $(V, E) \in \mathcal{G}$ is a finite directed graph.
- $Y : V \to \mathcal{B}_{types}$ is a total function that associates to each vertex its type. The function Y induces a partition on the set V, for instance the set $T = \{v \in V \mid Y(v) \in \{\mathsf{AT}, \mathsf{CT}\}\}$ denotes the set of tasks, while $D = \{v \in V \mid Y(v) \in \{\mathsf{DO}, \mathsf{DI}, \mathsf{DU}\}\}$ is the set of data objects.
- $X : E \to \{\mathsf{S}, \mathsf{A}\}$ is a total function that associate to each edge its type, where S denotes a sequence flow, and A is an association.
- A is a set of labels such that $\tau \notin A$ where τ denotes the silent action.
- $L : T \in A \cup \{\tau\}$ is a total function that associates a label to each task, eventually the silent one.
- $D : T \to \wp(DO \times DI \times DU)$ is the function that associates data objects to tasks.

The basic access functions are summarized in Tab. 3.11 and they are defined as follows: for any $a \in \mathcal{B}$ such that $a = (V, E, Y, X, A, L, D)$, $vertices(a) = V$, $edges(a) = E$. Function $tasks : \mathcal{B} \to \wp(\mathcal{U})$ returns the set of components in a model that are atomic or composite task; formally $tasks(a) = \{v \in V \mid Y(v) \in \{\mathsf{AT}, \mathsf{CT}\}\}$. Similarly, function $events : \mathcal{B} \to \wp(\mathcal{U})$ returns the set of components in a model that are events, namely $events(a) = \{v \in V \mid Y(v) \in \{\mathsf{SE}, \mathsf{IE}, \mathsf{EE}, \mathsf{TE}\}\}$; while function $gateways : \mathcal{B} \to \wp(\mathcal{U})$ returns the set of components in a model that are gateways, $gateways(a) = \{v \in V \mid Y(v) \in \{\mathsf{XG}, \mathsf{OG}, \mathsf{AG}, \mathsf{DG}\}$. Function $data\text{-}objects : \mathcal{B} \to \wp(\mathcal{U})$ determines the set of components in a model that are data objects, formally

$\mathit{data\text{-}objects}(a) = \{v \in V \mid Y(v) \in \{\mathsf{DO}, \mathsf{DI}, \mathsf{DU}\}\}$. Finally, function $\mathit{seq\text{-}flows} : \mathcal{B} \to \wp(\mathcal{U})$ returns the set of sequence flows in a model, $\mathit{seq\text{-}flows}(a) = \{e \in E \mid Y(v) = \mathsf{S}\}$, and function $\mathit{associations} : \mathcal{B} \to \wp(\mathcal{U})$ returns the set of associations in a model, $\mathit{associations}(a) = \{e \in E \mid Y(v) = \mathsf{A}\}$.

Table 3.11. Basic operations on BPMN nets.

Symbol	Function	Description	Ref
	$\mathit{vertices} : \mathcal{B} \to \wp(\mathcal{U})$	The underlying graph vertices.	*inline*
	$\mathit{edges} : \mathcal{B} \to \wp(\mathcal{U})$	Edges of the underlying graph.	*inline*
	$\mathit{tasks} : \mathcal{B} \to \wp(\mathcal{U})$	Vertices that are atomic or composite tasks.	*inline*
	$\mathit{events} : \mathcal{B} \to \wp(\mathcal{U})$	Vertices that are start events.	*inline*
	$\mathit{gateways} : \mathcal{B} \to \wp(\mathcal{U})$	Vertices that are exclusive gateways.	*inline*
	$\mathit{data\text{-}objects} : \mathcal{B} \to \wp(\mathcal{U})$	Vertices that are data objects.	*inline*
	$\mathit{seq\text{-}flows} : \mathcal{B} \to \wp(\mathcal{U})$	Edges that are sequence flows.	*inline*
	$\mathit{associations} : \mathcal{B} \to \wp(\mathcal{U})$	Edges that are associations.	*inline*
	$\mathit{vertex\text{-}type} : \mathcal{B} \times \mathcal{U} \to \mathcal{B}_{types}$	Vertex type.	Eq. 3.74
	$\mathit{edge\text{-}type} : \mathcal{B} \times \mathcal{U} \to \{\mathsf{AT}, \mathsf{CT}\}$	Edge type.	Eq. 3.75
	$\mathit{task\text{-}data} : \mathcal{B} \times \mathcal{U} \to \wp(\mathcal{U})$	Data objects of a task.	Eq. 3.76

Function $\mathit{vertex\text{-}type} : \mathcal{B} \times \mathcal{U} \to \mathcal{B}_{types}$ associates to each vertex in a model its type, its behaviour is reported in Eq. 3.74.

$$\begin{aligned} &\mathit{vertex\text{-}type} : \mathcal{B} \times \mathcal{U} \to \mathcal{B}_{types} \\ &\quad \forall a \in \mathcal{B} \,.\, u \in \mathcal{U} \,.\, a = (V, E, Y, X, A, L, D) \,. \\ &\qquad \mathit{vertex\text{-}type}(a, u) = \begin{cases} Y(u) & \text{if } u \in \mathit{vertices}(a) \\ \uparrow & \text{otherwise} \end{cases} \end{aligned} \tag{3.74}$$

Function $\mathit{edge\text{-}type} : \mathcal{B} \times \mathcal{U} \to \{\mathsf{AT}, \mathsf{CT}\}$ associates to each edge in a model its type, its behaviour is reported in Eq. 3.75.

$$\begin{aligned} &\mathit{edge\text{-}type} : \mathcal{B} \times \mathcal{U} \to \{\mathsf{AT}, \mathsf{CT}\} \\ &\quad \forall a \in \mathcal{B} \,.\, u \in \mathcal{U} \,.\, a = (V, E, Y, X, A, L, D) \,. \\ &\qquad \mathit{edge\text{-}type}(a, u) = \begin{cases} X(u) = x & \text{if } u \in \mathit{edges}(a) \\ \uparrow & \text{otherwise} \end{cases} \end{aligned} \tag{3.75}$$

Finally, function $\mathit{task\text{-}data} : \mathcal{B} \times \mathcal{U} \to \wp(\mathcal{U})$ associates to each task in a model its data objects, its behaviour is reported in Eq. 3.76.

$$\begin{aligned} &\mathit{task\text{-}data} : \mathcal{B} \times \mathcal{U} \to \wp(\mathcal{U}) \\ &\quad \forall a \in \mathcal{B} \,.\, \forall t \in \mathit{tasks}(a) \,.\, \forall u \in \mathcal{U} \,.\, \in \mathit{task\text{-}data}(a, t) \overset{\Delta}{\iff} \\ &\qquad a = (V, E, Y, X, A, L, D) \wedge \exists (u, t) \in \mathit{edges}(a) \,.\, u \in \mathit{data\text{-}objects}(a) \end{aligned} \tag{3.76}$$

$\diamond$

3.6 Summary and Concluding Remarks

This chapter provides a uniform overview about the design of graphical PMLs and their theoretical foundation. It discusses four graphical modeling languages from the most formal one. The chapter starts with a presentation and formalization of the PTNs language that is substantially self-modifying place transition nets from which other Petri nets variants can be obtained with simple syntactical restrictions. PTNs is more expressive than ordinary Petri nets, e.g. it is Turing-complete, anyway it is very limited in data representation and manipulation. The chapter continues by introducing the CPNs language. CPNs enhances basic Petri nets with a powerful inscription language that can be used to build complex expressions and data types. Finally, the chapter deals with less formal modeling languages, like YAWL and BPMN. Each of these languages is described using a similar approach: initially the graphical language elements are presented together with their syntax, then an informal interpretation of each language constructs is provided, followed by a mathematical formalization whenever possible depending on the original specification.

The languages presented here will be used in the following chapters for exemplifying the discussed concepts or proving certain properties. In particular, Petri nets will be applied in the next chapter for discussing the problems related to model verification and correction, while YAWL and BPMN will be used in Chap. 5 for exposing some issues about the promoted paradigm for process design.

4

Free Composition, Verification and Correction

A main step in any process design activity is the simulation and verification of the modeled processes, with the goal to discover the presence of particular errors and eventually rectify them. Workflow nets [75] are a variant of Petri nets that has been extensively adopted for the analysis and verification of business processes. Workflow nets entails a notion of soundness that has been introduced for distinguishing nets with a good run-time behaviour from those that will lead to an erroneous state during their execution. In this chapter the original notion of soundness is discussed and revised to ease the correction of errors found during the verification phase. Accordingly a refined version of the soundness check technique is described.

Very few verification tools provide useful hints about how to fix the found errors: end-users have to figure out themselves the solution by interpreting the often cryptic output of such tools. Model correction is rarely a trivial task: apparently independent problems can have a common cause and the correction of an error in one place can introduce new errors in other parts of the model.

The aim of this chapter is twofold. Firstly, it refines the notion of soundness and its related verification method for producing less redundant and more detailed error messages. Secondly, it introduces a novel technique, called Petri Nets Simulated Annealing (PNSA), used for streamlining process model correction. The idea behind the PNSA technique is searching models that are both structurally and semantically similar to the original one but containing few errors, taking advantage of the output of the refined soundness check method. The syntactical differences between a model produced by the PNSA technique and the original one can be offered as hints to rectify the existing problems.

The remainder of this chapter is organized as follows: Sec. 4.1 summarizes several research efforts about workflow nets, soundness, verification methods and the automated program repair problem. Sec. 4.2 describes the typical phases of a design activity to show where a correction tool can be used. Sec. 4.3 introduces workflow nets, while Sec. 4.4 and Sec. 4.5 discuss the notion of soundness and the related verification method, respectively. Sec. 4.6 introduces the automated model repair problem taking workflow nets as the reference formal language. Sec. 4.7 proposes the aforementioned PNSA technique. The technique is validated against a set of real process models redefined as workflow nets. The results of such preliminary validation are discussed in Sec. 4.8.

4.1 Related Work

Petri nets theory offers a wide range of methods for modeling and analyzing processes of various nature. Workflow nets is a formal language tailored for the modeling and verification of processes in the workflow management domain, obtained by adding some constraints to ordinary Petri nets. It appears for the first time in the seminal work of van der Aalst [72, 75] together with a definition of soundness and a set of transformation rules to evolve a net preserving its correctness.

Over the years several extensions of Workflow Nets, different notions of soundness and related verification methods have been proposed. A survey of the most relevant soundness criteria can be found in the work of Weske [7] and van der Aalst et al. [76]. The former discusses classical soundness, weak soundness [77], relaxed soundness [78] and lazy soundness [79] pointing out how they are related. The later surveys additional soundness formalizations and their related decidability issues for classical Workflow nets and other extensions.

Model checking, automated theorem proving and verification tools are consolidated research topics well documented in scientific literature. A less considered topic regards how to help the end-user in formulating a solution for solving the problems found in a model. In the context of PMLs, in [80] the authors describe a technique for automatically fixing certain types of data anomalies that can occur in process models, while in [81] an approach is presented to compute the edit operations required to correct a faulty service in order to interact in a choreography without deadlocks.

The problem addressed in this chapter shares many commonalities with the *Automated Program Repair (APR)* problem, namely how to automatically fix bugs inside programs written in a general-purpose language. The APR problem can be stated as follows: given a program, a set of positive tests and at least a failed one proving the presence of a bug, produce a patch that fixes the error in question. Test cases can be provided in advance or generated on demand from formal specifications or ad-hoc procedures. Arcuri [82], Forrest et al. [83] and Wilkerson et al. [84] have recently tackled the APR problem with co-evolutionary genetic programming (GP) techniques. Genetic and Evolutionary Computations (GECs) encompass a wide range of methods for solving complex optimization and combinatorial problems by means of basic algorithmic operations that mimic natural phenomena. GP in particular identifies a subclass of evolutionary computations focused on problems that require solutions with a certain structure, like trees and graphs, as it is usually necessary in the synthesis of computer programs and circuits.

The main steps of the APR techniques proposed in [82–84] can be broadly summarized as follows: (1) localize the faulty code using the failed tests, then (2) evolve new code fragments with GP techniques in an attempt to obtain a program similar to the original one, but able to pass all the given tests. If such program is found, (3) simplify it and use the remaining differences to create a patch. These techniques cannot guarantee neither that a solution is found, nor that a solution introduces subtle bugs not contemplated by the available tests. Despite these limitations, they are able to produce good fixes for preexisting programs written in the C programming language [83, 84].

Two reasonable assumptions heavily influence the effectiveness of an APR technique [82, 83]: (1) program defects are local and (2) optimal solutions are structurally close to the original program. As a consequence, only small fragments of the original program need to be altered, limiting the search space and the chance of corrupting parts not covered by any test. With these assumptions, traditional GP techniques may not be the best way to tackle the APR problem: for instance, in GP applications the first generation of programs is randomly sampled to cover as much as possible the entire search space, while in APR the first generation is composed of nearly identical individuals where variety should be artificially enforced to make GP works. High variety is preferable for avoiding premature convergence to unsatisfactory solutions, but at the same time it reduces the probability to find optimal solutions close to the original program, hence the need for a further simplification phase when a solution is found. At the end, the confidence about the correctness of a solution is inversely proportional to its distance from the original program; consequently, strange solutions are more likely to be discarded by the end-user even if they are correct.

The PNSA technique presented in this chapter can be seen as an APR technique tailored for executable process models. Nevertheless, in light of the above considerations, PNSA attempts a different approach that aims to systematically search near to the original model, before exploring the surrounding solutions, without growing large populations. PNSA is mainly inspired by the work of Suman [85] and Smith et al. [86] about dominance-based *Multi-Objective Simulated Annealing (MOSA)*. The key idea of these MOSA methods is to estimate the gap between the current solution and the unknown optimal solutions of the Pareto-front, and use it as a single fitness function to be minimized. Optimal solutions are approximated by a finite set of mutually non-dominated candidate solutions found so far during the search. The general goal of the method is to move towards the Pareto-front encouraging the diversification of the candidate solutions. It worths to be stressed that PNSA is inspired by, but not based on, these MOSA methods because it remains a GP technique: as such, it does not fall in the category of MOSA applications.

◊

4.2 The Free Composition Paradigm

A process model is the final result of a design activity carried out by the end-user by hand or more often using specific software editors [66,69,87]. Excluding trivial cases, a design activity is a sequence of refining steps, iterated multiple times until predefined requirements and quality criteria are met. At each step multiple design alternatives can be evaluated and a new version of the model is crafted by applying a sequence of changes to the existing one.

The primary goals of an editor are to ease model manipulation, track versions, and enforce basic syntactical constraints; for instance, in Petri net construction an editor can deny the connection of two nodes of the same type. Another important feature for model comprehension and testing purposes is the simulation and the actual execution of a process model. Formal verification methods are also widely exploited for detecting potential design flaws or for guaranteeing the presence of desired semantic properties [88].

The end-user can take advantage of the information collected with different simulation and verification techniques to identify model problems. Generally it is up to the user to understand the output produced by these tools and figure out how to solve the problems respecting the imposed language constraints. Find the right fix is not always a trivial task: apparently independent problems may have a common cause and the correction of an error in one part may introduce new errors in other parts of the model. As if this was not enough, the new introduced errors can escape outside the range of errors that can be automatically detected.

A typical design session can be seen as a sequence of steps that transform an existing model into a different but correlated model. A generic *design step* can be ideally decomposed in five distinct phases exemplified in Fig. 4.1. Initially, (1) the end-user conceptualizes a new model in an attempt to include the missing features, then (2) the existing model is modified following the construction rules imposed by the modeling language. The obtained model does not necessarily match the desired one, hence (3) the end-user exploits the available simulation and verification tools to check the new added parts. If the model is not consistent, (4) the end-user tries to fix it with the collected information, usually reiterating the previous phases until all problems are solved. The resulting model is finally (5) evaluated against the original intention and rejected or accepted. When it is accepted, the model becomes a new starting point for the next design step.

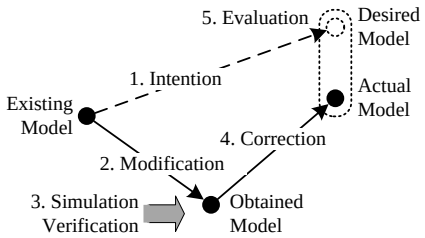


Fig. 4.1. A generic design step.

The described decomposition does not exclude simpler interactions: the obtained model can be exactly the desired one and the applied changes do not necessarily introduce new errors. The aim of this chapter is to propose a method for driving the user during the correction phase by automatically determining a set of possible corrections.

4.3 Workflow Nets

The *Workflow Nets (WFNs)* [75] language is a Petri nets variant tailored for designing processes in the context of workflow management systems. A *workflow net* is nothing more than an ordinary net with further syntactical restrictions and a particular halting semantics: the net shall have a single start place with no incoming arcs, a single end place with no outgoing arcs, and any other component shall lay on a path between these two places. Unlike usual Petri nets, the execution of a workflow net halts when at least one token reaches the end place. These new rules are introduced for several reasons, one of them is to ease the hierarchical composition of workflow nets starting from existing ones.

Definition 4.1 (Original Workflow Nets). In [75] a Petri net is defined as a tuple (P, T, F) made from a finite number of places and transitions contained respectively in two sets $P, T \subseteq \mathcal{U}$ such that $P \cap T = \varnothing$ and $F \subseteq (P \times T) \cup (T \times P)$. The set of all Petri nets yet described is denoted by $\mathcal{N}_{origin}$:

$$\begin{aligned} \forall z \in \mathcal{U} \,.\, z \in \mathcal{N}_{origin} \triangleq\!\!\iff \exists P, T, F \in \mathcal{U} \,.\, (P, T, F) = z \wedge \\ \exists n \in \mathbb{N} \,.\, |P| + |T| < n \;\wedge\; P \cap T = \varnothing \;\wedge\; F \subseteq (P \times T) \cup (T \times P) \end{aligned} \tag{4.1}$$

A workflow net [75] is then defined as a Petri net $z \in \mathcal{N}_{origin}$ having a initial place p_s without incoming arcs, a final place p_e without outgoing arcs and the remaining components in a path from p_s to p_e. The set of all workflow nets $\mathcal{W}_{origin}$ is formally defined as follows:

$$\begin{aligned} \forall z \in \mathcal{U} \,.\, z \in \mathcal{W}_{origin} \triangleq\!\!\iff \exists P, T, F \in \mathcal{U} \,. \\ z = (P, T, F) \in \mathcal{N}_{origin} \wedge \\ \exists p_s \in P \,.\, \forall t \in T \,.\, (t, p_s) \notin F \wedge \\ \exists p_e \in P \,.\, \forall t \in T \,.\, (p_e, t) \notin F \wedge \\ \forall u \in P \cup T \,.\, u \in \mathcal{C}(p_s) \end{aligned} \tag{4.2}$$

where $\mathcal{C}(x)$ represents the set of all the net elements reachable from x that can be recursively defined as $\forall u \in \mathcal{U} \,.\, u \in \mathcal{C}(x) \iff u = x \vee \exists v \in \mathcal{C}(x) \,.\, (v, u) \in F$.

Following the usual convention, the fields of a workflow net will be denoted by the functions $pls : \mathcal{W}_{origin} \to \mathcal{U}$, $trs : \mathcal{W}_{origin} \to \mathcal{U}$ and $\mathit{flow\text{-}relation} : \mathcal{W}_{origin} \to \mathcal{U} \times \mathcal{U}$, such that for all $a = (P, T, F) \in \mathcal{W}_{origin}$, $pls(a) = P$, $trs(a) = T$ and $\mathit{flow\text{-}relation}(a) = F$. Both the initial place p_s and final place p_e of a workflow net $a \in \mathcal{W}_{origin}$ are unique and they will be denoted by the functions $start : \mathcal{W}_{origin} \to \mathcal{U}$ and $end : \mathcal{W}_{origin} \to \mathcal{U}$, respectively. Without losing generality, the initial marking of a workflow net $a \in \mathcal{W}_{origin}$ is assumed to be a single token in the $start(a)$ place.

A workflow net $a \in \mathcal{W}_{origin}$ starts its execution when a token is put in the initial place and it terminates when a token appears in the final place [75]. Clearly, the halting condition of workflow nets differs from the one given for PTNs: the execution terminates when no other transition is enabled or when at least one token is stored in the end place. This behavior can be obtained in two ways: the first one is modifying the PTNs interpreter in order to check the end place condition, the second one is traducing each workflow nets to a compatible net before its

execution. Here, the latter solution is adopted, hence the formal semantics of the original workflow nets is given in PTNs through the translating function $translate : \mathcal{W}_{origin} \to \mathcal{N}$ defined in Eq. 4.3.

$$\begin{aligned} &translate : \mathcal{W}_{origin} \to \mathcal{N} \\ &\quad \forall a \in \mathcal{W}_{origin} \,.\, \forall b \in \mathcal{N} \,.\, b = translate(a) \iff \\ &\qquad places(b) = \{(p, \omega) \mid p \in pls(a)\} \;\wedge \\ &\qquad transitions(b) = \{(t, \tau) \mid t \in trs(a)\} \;\wedge \\ &\qquad arcs(b) = \{(x, y, 1, \theta) \mid (x, y) \in \textit{flow-relation}(a)\} \cup \\ &\qquad\qquad \cup \{(end(a), t, 2, end(a)) \mid t \in trs(a)\} \end{aligned} \tag{4.3}$$

For every workflow net, the translating function returns a net identical to the original one except for the final place that is connected to every existing transition with an inhibitor arc: when a token is stored in such place, no other transition can be enabled, hence the execution is complete.

Not surprisingly, Petri nets literature contains several extensions of the original concept of workflow nets [76]. Here, it is natural to define *Workflow Nets (WFNs)* as a restriction of the already discussed PTNs language in order to inherit its features like actions or place capacities.

Definition 4.2 (Halt Place). A place $p \in pls(a)$ of a net $a \in \mathcal{N}$ is said to be an *halt place* if and only if it is connected to every transition $t \in trs(a)$ with an inhibitor arc. Function $\textit{halt-place} : \mathcal{N} \times \mathcal{U} \to \mathbb{B}$ defined in Eq. 4.4 is used to capture such notion.

$$\begin{aligned} &\textit{halt-place} : \mathcal{N} \times \mathcal{U} \to \mathbb{B} \;\; is \\ &\quad \forall a \in \mathcal{N} \,.\, \forall p \in pls(a) \,.\, \textit{halt-place}(a, p) \iff \\ &\qquad \forall t \in trs(a) \,.\, \exists w \in \mathbb{N} \,.\, (p, t, w, p) \in arcs(a) \wedge w > 1 \end{aligned} \tag{4.4}$$

For not cluttering the graphical representation of a net, an halt place will be denoted with a double circled place, as happens for p_h in Fig. 4.2. In this manner the related inhibitor arcs can be left implicit. The intuitive meaning is exemplified in the same figure: replace the halt place with a regular one and connect it to every transition of the net through an outgoing inhibitor arc.

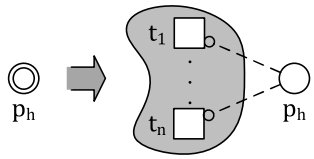


Fig. 4.2. Halt place notation and its intuitive meaning.

Definition 4.3 (Workflow Nets). A net $a \in \mathcal{N}$ is a *workflow net* if and only if (1) there exists a unique start place $p_s \in pls(a)$ without incoming arcs, (2) there exists a unique end place $p_e \in pls(a)$ without outgoing arcs that is also an halt place, and (3) every vertex of the net lays on a path from p_s to p_e. The set of all workflow nets is denoted by $\mathcal{W}$ and formally defined as follows:

$$\begin{aligned} &\forall a \in \mathcal{N} \,.\, a \in \mathcal{W} \iff \\ &\quad \exists p_s \in pls(a) \,.\, \textit{pre-set}(a, p_s) = \varnothing \wedge \\ &\quad \exists p_e \in pls(a) \,.\, \textit{post-set}(a, p_e) = \varnothing \wedge \textit{halt-place}(a, p_e) \wedge \\ &\quad \forall v \in vertices(a) \,.\, v \in components(a, p_s) \end{aligned} \tag{4.5}$$

The initial place p_s and the final place p_e of a net $a \in \mathcal{W}$ are unique: for contradiction, if there exists $a \in \mathcal{W}$ with two different start places $p_{s1}, p_{s2} \in \mathit{pls}(a)$, such that $p_{s1} \neq p_{s2}$, $\mathit{pre\text{-}set}(a, p_{s1}) = \varnothing$ and $\mathit{pre\text{-}set}(a, p_{s2}) = \varnothing$, then by Def. 4.3 it follows $p_{s2} \in \mathit{components}(a, p_{s1})$ and vice versa. But $p_{s2} \in \mathit{components}(a, p_{s1})$ only if p_{s2} reachable with an incoming arc, hence $\mathit{pre\text{-}set}(a, p_{s2}) \neq \varnothing$, or $p_{s2} = p_{s1}$. The same holds for p_{s1} if $p_{s1} \in \mathit{components}(a, p_{s2})$. Initial and final places are denoted by the functions $\mathit{start} : \mathcal{W} \to \mathcal{U}$ and $\mathit{end} : \mathcal{W} \to \mathcal{U}$, respectively.

A workflow net $a \in \mathcal{W}$ is a standard net, because every transition has a non-empty pre-set, hence it cannot make progress without an initial marking. By default the initial marking of a workflow net $a \in \mathcal{W}$ is a single token in the start place, i.e. $\mathit{initial\text{-}marking}(a) \triangleq \{\mathit{start}(a) \mapsto 1\}$. The set of *marked workflow nets* $\mathcal{W}^\bullet$ is defined in the same way of $\mathcal{N}^\bullet$ as a pair formed by a net and a valid marking, i.e. $\mathcal{W}^\bullet \triangleq \{\ (a, q) \mid a \in \mathcal{W} \wedge q \in \mathit{markings}(a)\}$.

In similar way the final marking is defined as $\mathit{final\text{-}marking}(a) \triangleq \{\mathit{end}(a) \mapsto 1\}$ and any marking that covers the final one is called *halt marking*. Formally,

$$\begin{aligned} &\mathit{halt\text{-}marking} : \mathcal{W} \times \mathcal{Q} \rightharpoonup \mathbb{B} \ \mathit{is} \\ &\quad \forall a \in \mathcal{W} \,.\, \forall q \in \mathcal{Q} \,. \\ &\qquad \mathit{halt\text{-}marking}(a, q) \triangleq\!\!\iff q \in \mathit{markings}(a) \wedge q \geq \mathit{final\text{-}marking}(a) \end{aligned} \tag{4.6}$$

The PTNs model in Fig. 4.3.a is an example of workflow net; its canonical interpretation is shown in Fig. 4.3.b where the halt place has been replaced by a set of inhibitor arcs that enforce the halting semantics stated for workflow nets. Clearly, the use of such inhibitor arcs does not exclude the existence of nets that terminate correctly without them but this is not the norm, e.g. no token is left behind in the net of Fig. 4.3.b when one is stored in p_5, while this is not the case for the net in Fig. 4.3.c where t_3 can fire indefinitely unless inhibitor arcs are added.

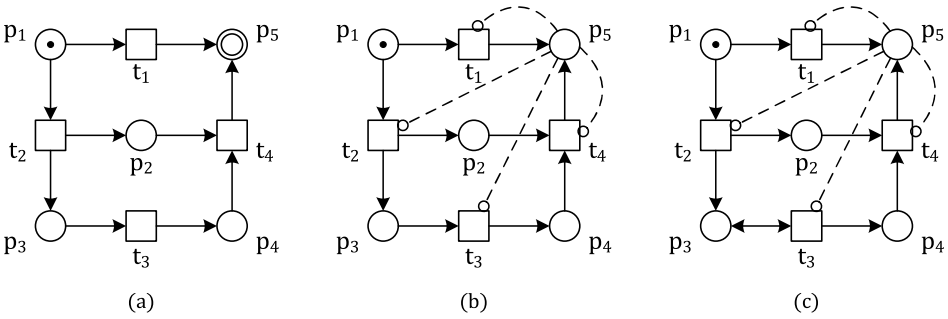


Fig. 4.3. (a) A workflow net with a special notation for the halt place. (b) The same net where the halt place has been translated to a regular place with inhibitor arcs. (c) A slightly different net that needs an halt place to comply with WFNs semantics.

The choice of adopting a single start place and a default marking in WFNs is reasonable since it does not reduce the language expressiveness, as shown by the following proposition.

Proposition 4.4 (Single Start Place). For every marked net $a \in \mathcal{N}^\bullet$, there exists a behaviorally equivalent marked net $b \in \mathcal{N}^\bullet$ belonging to the same Petri nets class, that has a place with an empty pre-set and a single token in it. In particular, the behavior of b is identical to the original one after a finite yet unavoidable sequence of silent initial transitions.

$$\begin{aligned}\forall a \in \mathcal{N}^\bullet \,.\, \exists b \in \mathcal{N} \,.\, \exists x \in \mathit{pls}(b) \,.\, \exists q_s \in \mathit{markings}(b) \,. \\ \mathit{pre\text{-}set}(b, x) = \varnothing \wedge \\ q_s(x) = 1 \wedge \forall y \in \mathit{pls}(b) \,.\, y \neq x \Rightarrow q(y) = 0 \wedge \\ \exists r \in \rho(b, q_s) \,.\, (b, r) \simeq_g a\end{aligned} \tag{4.7}$$

Workflow Nets Classification

Several classes of workflow nets can be defined by considering the presence of special arcs inside them. In particular, in order to make such classification the following function $\mathit{ptn} : \mathcal{W} \to \mathcal{N}$ is initially used, which transforms a workflow net $a \in \mathcal{W}$ into an equivalent PTNs net $b \in \mathcal{N}$ in which the inhibitor arcs between any transition and the end place have been removed, and then the presence of remaining special arcs is evaluated.

$$\begin{aligned}&\mathit{ptn} : \mathcal{W} \to \mathcal{N} \;\; \mathit{is} \\ &\quad \forall a \in \mathcal{W} \,.\, b = \mathit{ptn}(a) \stackrel{\triangle}{\Longleftrightarrow} \\ &\qquad b \;\; \mathit{same\ as}\ a \;\; \mathit{except} \\ &\qquad \mathit{arcs}(b) = \mathit{arcs}(a) \setminus \{\, (\mathit{end}(a), t, w, \mathit{end}(a)) \mid t \in \mathit{trs}(a) \wedge w \in \mathbb{N} \setminus \{1\} \}\end{aligned} \tag{4.8}$$

Given such transformation function, the following classes of workflow nets can be defined on the basis of the presence the special arcs presented in Sec. 3.2.4. In particular, $\mathcal{W}_{ext}$ is the class of workflow nets containing reset and inhibitor arcs, $\mathcal{W}_{reset}$ is the class of workflow nets containing only reset arcs, and $\mathcal{W}_{basic}$ is the class of workflow nets containing only basic arcs.

$$\mathcal{W}_{ext} = \{a \in \mathcal{W} \mid \mathit{ptn}(a) \in \mathcal{N}_{reset} \cup \mathcal{N}_{inhibitor}\} \tag{4.9}$$

$$\mathcal{W}_{reset} = \{a \in \mathcal{W} \mid \mathit{ptn}(a) \in \mathcal{N}_{reset}\} \tag{4.10}$$

$$\mathcal{W}_{basic} = \{a \in \mathcal{W} \mid \mathit{ptn}(a) \in \mathcal{N}_{basic}\} \tag{4.11}$$

The following function allows one to transform a workflow net $a \in \mathcal{W}$ into an equivalent PTNs net $b \in \mathcal{N}$, but differently from function $\mathit{translate} : \mathcal{W} \to \mathcal{N}$ defined in Eq. 4.8, in this case no inhibitor arcs are used:

$$\begin{aligned}&\mathit{basic\text{-}translate} : \mathcal{W} \to \mathcal{N} \;\; \mathit{is} \\ &\quad \forall a \in \mathcal{W} \,.\, \forall b \in \mathcal{N} \,.\, b = \mathit{basic\text{-}translate}(a) \stackrel{\triangle}{\Longleftrightarrow} \\ &\qquad b \;\; \mathit{same\ as}\ a \;\; \mathit{except} \\ &\qquad \mathit{pls}(b) = \mathit{pls}(a) \cup \{(p_h, \omega)\} \wedge \\ &\qquad \mathit{arcs}(b) = \mathit{arcs}(\mathit{ptn}(a)) \cup \\ &\qquad\quad \cup \{\, (p_h, t, 1, \theta) \mid t \in \mathit{trs}(a) \wedge t \notin \mathit{post\text{-}set}(a, \mathit{start}(a)) \,\} \cup \\ &\qquad\quad \cup \{\, (t, p_h, 1, \theta) \mid t \in \mathit{trs}(a) \wedge t \notin \mathit{pre\text{-}set}(a, \mathit{end}(a)) \,\} \setminus\end{aligned} \tag{4.12}$$

where p_h is a new place identifier not in $pls(a)$ and unspecified attributes like actions and capacities remain the same.

The idea behind this translating function is the following: every transition directly connected with the initial place but not with the final one stores a single token in the added place p_h, every transition directly connected with the final place but not with the initial one removes the token in p_h and every transition neither in the post-set of the initial place, nor in the pre-set of the final place is disabled if p_h is empty and left the token in it if it fires.

This translating function does not alter the Petri nets class of the given workflow net. The resulting net is not necessary a workflow net but this is true when the net is non-trivial, i.e. there exists at least one transition in the post-set of the initial place not in the pre-set of the final place and vice versa.

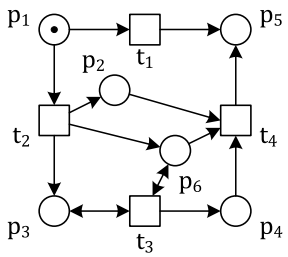


Fig. 4.4. The result of the preserving translation applied to the workflow net in Fig. 4.3.c

Definition 4.5. A workflow net $a \in \mathcal{W}$ is non-trivial if it contains at least a transition which is contained in the post-set of the initial place, but not in the pre-set of the the final place, and vice versa. This condition is formally specified as follows:

$$\begin{aligned} &\forall a \in \mathcal{W} \,.\, \textit{non-trivial}(a) \stackrel{\triangle}{\Longleftrightarrow} \\ &\quad \exists x, y \in \textit{trs}(a) \,. \\ &\qquad x \notin \textit{post-set}(a, \textit{start}(a)) \ \wedge \ y \notin \textit{pre-set}(a, \textit{end}(a)) \end{aligned} \tag{4.13}$$

Proposition 4.6 (Non-Trivial Workflow Nets). Given a workflow net $a \in \mathcal{W}$, if a is non-trivial, then the net $b = \textit{basic-translate}(a)$ is also a workflow net.

$$\forall a \in \mathcal{W} \,.\, \textit{non-trivial}(a) \Rightarrow \textit{basic-translate}(a) \in \mathcal{W} \tag{4.14}$$

To make a workflow net non-trivial it is sufficient to add a new initial place followed by a silent transition connected to the original initial place.

Extended Workflow Nets presented in [76] can be translated in workflow nets given here: nets with inhibitor arcs connected to a place in the preset are excluded, nets with reset arcs connected to a place in the preset need to be transformed.

4.4 The Notion of Soundness

This section introduces the notion of classical soundness [75] for WFNs models. Roughly speaking a workflow net is *sound* when it exposes a good run-time behavior. For classical soundness this is translated in three properties: a workflow net is sound if (1) there is always an opportunity to complete the execution, (2) every terminating execution completes properly with one token in the final place, and (3) every transition has at least one chance to fire.

Surprisingly, in Petri nets literature there is no consensus about the formal definition of classical soundness [75], hence this section introduces one of the existing variants and shows what not works with the others.

Definition 4.7 (Option to Complete). A workflow net $a \in \mathcal{W}$ has an *option to complete* if and only if for every marking reachable from the initial one, at least one token can reach the final place. The set of workflow nets with an option to complete is denoted by $\mathcal{W}_{opt} \subseteq \mathcal{W}$ and formally defined as follows:

$$\begin{aligned} &\forall a \in \mathcal{W} \,.\, a \in \mathcal{W}_{opt} \stackrel{\triangle}{\Longleftrightarrow} \\ &\quad \forall q \in \rho(a, qs(a)) \,.\, \exists r \in \rho(a, q) \,.\, r \geq qe(a) \end{aligned} \tag{4.15}$$

Definition 4.8 (Proper Completion). A workflow net $a \in \mathcal{W}$ *completes properly* if and only if every reachable halting state is the final state with a single token in the final place. The set of workflow nets with proper completion is denoted by $\mathcal{W}_{pro} \subseteq \mathcal{W}$ and formally defined as follows:

$$\begin{aligned} &\forall a \in \mathcal{W} \,.\, a \in \mathcal{W}_{pro} \stackrel{\triangle}{\Longleftrightarrow} \\ &\quad \forall q \in \rho(a, qs(a)) \,.\, q \geq qe(a) \Rightarrow q = qe(a) \end{aligned} \tag{4.16}$$

Definition 4.9 (No Dead Transitions). A workflow net $a \in \mathcal{W}$ has *no dead transitions* if and only if every transition can fire at least in one run. The set of workflow nets with no dead transitions is denoted by $\mathcal{W}_{nod} \subseteq \mathcal{W}$ and formally defined as follows:

$$\begin{aligned} &\forall a \in \mathcal{W} \,.\, a \in \mathcal{W}_{nod} \stackrel{\triangle}{\Longleftrightarrow} \\ &\quad \forall t \in trs(a) \,.\, \exists q \in \rho(a, qs(a)) \,.\, t \in \textit{enabled-set}(a, q) \end{aligned} \tag{4.17}$$

Definition 4.10 (Classical Soundness). A workflow net is *sound* if and only if it has at the same time an option to complete, proper completion and no dead transitions. The set of sound workflow nets is denoted by $\mathcal{W}_{sound} \subseteq \mathcal{W}$ and formally defined as follows:

$$\mathcal{W}_{sound} = \mathcal{W}_{opt} \cap \mathcal{W}_{pro} \cap \mathcal{W}_{nod} \tag{4.18}$$

To provide evidence that soundness properties are well defined it can be useful to prove the following assumptions:

1. Neither $\mathcal{W}_{sound}$ nor its complement $\mathcal{W}^c_{sound}$ are empty.
2. No one of the defined properties $\mathcal{W}_{opt}$, $\mathcal{W}_{pro}$ and $\mathcal{W}_{nod}$ are empty.
3. For all $X, Y \in \{\mathcal{W}_{opt}, \mathcal{W}_{pro}, \mathcal{W}_{nod}\}$, it holds that $X \cap Y \neq \varnothing$, i.e. no intersection of properties is empty.
4. The defined properties $\mathcal{W}_{opt}$, $\mathcal{W}_{pro}$ and $\mathcal{W}_{nod}$ are orthogonal or equivalently not redundant: a set of properties $\{X_i\}_{i=1}^n$ is *redundant* if and only if there exists a property that can be derived from another one or a combination of two or more of the remaining ones.

$$\{X_i\}_{i=1}^n \textit{ redundant} \iff \exists j \in [1,n] \,.\, \forall u \in \mathcal{U} \,.\, \left(u \in \bigcap_{i \neq j}^n X_i \Rightarrow u \in X_j \right)$$

Note that if for some non-empty set of indices $I \subseteq [1,n]$, it can be proven that for every $u \in \mathcal{U}$ holds $u \in \bigcap_{i \in I} X_i \Rightarrow u \in X_j$, then, relaxing the preconditions, for all $u \in \mathcal{U}$ follows $u \in \bigcap_{i \neq j} X_i \Rightarrow u \in X_j$.

The rationale behind such requirements is rooted on the role of workflow nets and soundness in real design activities: WFNs is not intended to be an end-user modeling language for real-world processes [76] because it lacks many essential features, like constructs to define and manage data. As a consequence, to apply any soundness check, a real model needs to be mapped on WFNs, abstracting away some details considered irrelevant for the analysis. Due to the abstraction process, a successful soundness check cannot exclude the presence of other kinds of errors. On the contrary, when the soundness check fails, it reveals one or more errors that can be directly related to the original model. For this reason it is better to have different orthogonal properties, so that each of them can spot a particular kind of error offering a better feedback to the end-user that wants to know what is wrong in order to correct it.

To prove that the soundness properties and their combinations are not empty it is sufficient to exhibit a set of eight workflow nets $Z = \{z_i\}_{i=1}^8$ each one representing a particular intersection of properties. Some of these nets can also be used as a counter example to prove that the properties are not redundant: for each property $X \in \{\mathcal{W}_{opt}, \mathcal{W}_{pro}, \mathcal{W}_{nod}\}$ it is sufficient to exhibit a net $z \in Z$ that has all the soundness properties except X.

Proposition 4.11 (Well Defined Soundness). There exists a representative workflow net for each combination of zero or more soundness requirements $\mathcal{W}_{opt}$, $\mathcal{W}_{pro}$ and $\mathcal{W}_{nod}$ formally stated in Eq. 4.15, Eq. 4.16 and Eq. 4.17, respectively. The given requirements are also orthogonal or equivalently not redundant.

Proof. The set of workflow nets $Z = \{z_i\}_{i=1}^8$ used in this proof is depicted in Fig. 4.5. All the given nets are necessary to prove that the properties and their intersections are not empty, while the nets z_2, z_3 and z_4 are sufficient to prove orthogonality.

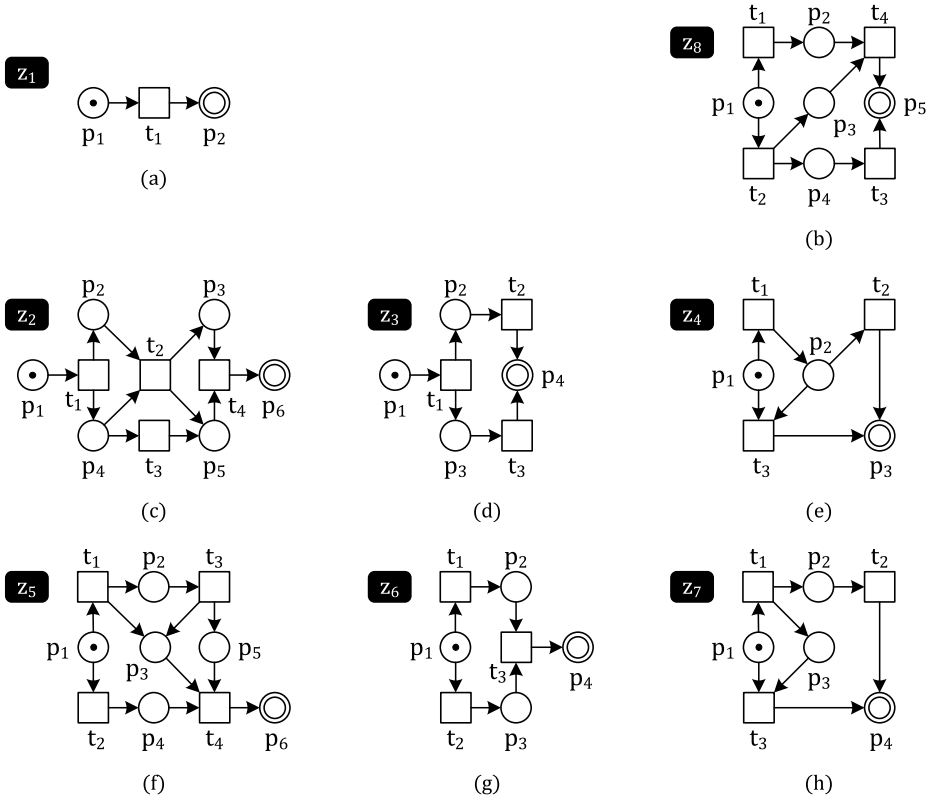


Fig. 4.5. All the nets in this figure are ordinary workflow nets, namely $\{z_i\}_{i=1}^{8} \subseteq \mathcal{W}_{basic}$. (a) The net z_1 is sound, $z_1 \in \mathcal{W}_{sound}$. (b) The net z_8 has no option to complete, improper completion and dead transitions, $z_8 \notin \mathcal{W}_{opt} \cup \mathcal{W}_{pro} \cup \mathcal{W}_{nod}$. (c) The net z_2 has no option to complete, $z_2 \in \mathcal{W}_{pro} \cap \mathcal{W}_{nod}$ but $z_2 \notin \mathcal{W}_{opt}$. (d) The net z_3 has improper completion, $z_3 \in \mathcal{W}_{opt} \cap \mathcal{W}_{nod}$ but $z_3 \notin \mathcal{W}_{pro}$. (e) The net z_4 has dead transitions, $z_4 \in \mathcal{W}_{opt} \cap \mathcal{W}_{pro}$ but $z_4 \notin \mathcal{W}_{nod}$. (f) The net z_5 has no option to complete and improper completion, $z_5 \notin \mathcal{W}_{opt} \cup \mathcal{W}_{pro}$ but $z_5 \in \mathcal{W}_{nod}$. (g) The net z_6 has no option to complete and dead transitions, $z_6 \notin \mathcal{W}_{opt} \cup \mathcal{W}_{nod}$ but $z_6 \in \mathcal{W}_{pro}$. (h) The net z_7 has improper completion and dead transitions, $z_7 \notin \mathcal{W}_{pro} \cup \mathcal{W}_{nod}$ but $z_7 \in \mathcal{W}_{opt}$.

z_1 The workflow net $z_1 \in \mathcal{W}_{basic}$ in Fig. 4.5.a has only two states $q_s = qs(z_1)$ and $q_e = qe(z_1)$ and the final one q_e is reachable from q_s firing t_1. These two states are the only reachable states from q_s; hence, there is always an option to complete. A proper completion is always guaranteed since the only halting state is the final one q_e. Clearly, t_1 cannot be a dead transition, as a consequence $z_1 \in \mathcal{W}_{sound}$.

z_2 The workflow net $z_2 \in \mathcal{W}_{basic}$ in Fig. 4.5.c can run following at most two paths $\langle p_1 \rangle \xrightarrow{t_1} \langle p_2 + p_4 \rangle \xrightarrow{t_2} \langle p_3 + p_5 \rangle \xrightarrow{t_4} \langle p_6 \rangle$ and from $\langle p_2 + p_4 \rangle$ it can reach $\langle p_2 + p_5 \rangle$ firing t_3. (1) The state $\langle p_2 + p_5 \rangle$ is an halting state and does not cover the final one $\langle p_6 \rangle$; hence, there is a no option to complete, $z_2 \notin \mathcal{W}_{opt}$. (2) The net has proper completion because the only halting state that covers $\langle p_6 \rangle$ is $\langle p_6 \rangle$ itself, $z_2 \in \mathcal{W}_{pro}$. (3) The net has no dead transitions because all the transitions appear in at least one of the two possible runs, $z_2 \in \mathcal{W}_{nod}$.

z_3 The workflow net $z_3 \in \mathcal{W}_{basic}$ in Fig. 4.5.d can directly reach, from the initial state $qs(z_3) = \langle p_1 \rangle$, only $\langle p_2 + p_3 \rangle$ by firing t_1, then it can reach $\langle p_3 + p_4 \rangle$ and $\langle p_2 + p_4 \rangle$ by firing t_2 or t_3, respectively. (1) Clearly, from the initial state every reachable state can reach $\langle p_3 + p_4 \rangle$ or $\langle p_2 + p_4 \rangle$ that both cover the final state $qe(z_3) = \langle p_4 \rangle$, hence there is always an option to complete, $z_3 \in \mathcal{W}_{opt}$. (2) The run $\langle p_1 \rangle \xrightarrow{t_1} \langle p_2 + p_3 \rangle \xrightarrow{t_2} \langle p_3 + p_4 \rangle$ terminates in an halting state that covers the final one $\langle p_4 \rangle$ but it is not exactly $\langle p_4 \rangle$, hence there is at least an improper completion, $z_3 \notin \mathcal{W}_{pro}$. (3) The net has no dead transitions because all the transitions appear in at least one of the two possible runs, $z_3 \in \mathcal{W}_{nod}$.

z_4 The workflow net $z_4 \in \mathcal{W}_{basic}$ in Fig. 4.5.e has only one possible run $\langle p_1 \rangle \xrightarrow{t_1} \langle p_2 \rangle \xrightarrow{t_2} \langle p_3 \rangle$, hence (1) for every state reachable from the initial one $qs(z_4) = \langle p_1 \rangle$ there is always an option to complete, $z_4 \in \mathcal{W}_{opt}$ and (2) the unique halting state is exactly the final marking $qe(z_4) = \langle p_3 \rangle$, so $z_4 \in \mathcal{W}_{pro}$. (3) The transition t_3 is dead because it never occurs in a run, hence $z_4 \notin \mathcal{W}_{nod}$.

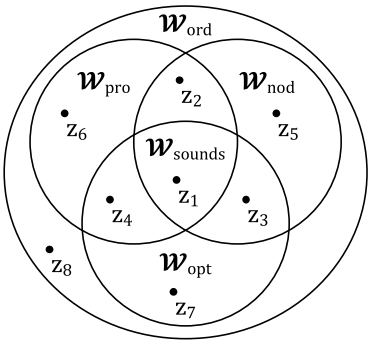
z_5 The workflow net $z_5 \in \mathcal{W}_{basic}$ in Fig. 4.5.f can evolve in two different ways from the initial state $qs(z_5) = \langle p_1 \rangle$: if it fires t_2 then it gets stuck in $\langle p_4 \rangle$, or if it fires t_1, it can cover the final state $qe(z_5) = \langle p_6 \rangle$ running $\langle p_1 \rangle \xrightarrow{t_1} \langle p_2 + p_3 \rangle \xrightarrow{t_3} \langle 2p_3 + p_5 \rangle \xrightarrow{t_4} \langle p_3 + p_6 \rangle$. (1) In the first case, the workflow net has no option to complete, so $z_5 \notin \mathcal{W}_{opt}$. (2) In the second one, it has an improper completion because when a token reaches p_6 there is a token left in the net in p_3, it follows $z_5 \notin \mathcal{W}_{pro}$. (3) All transitions occur in at least in one run, then $z_5 \in \mathcal{W}_{nod}$.

z_6 The workflow net $z_6 \in \mathcal{W}_{basic}$ in Fig. 4.5.g can reach, from the initial state $qs(z_6) = \langle p_1 \rangle$, state $\langle p_2 \rangle$ by firing t_1 or state $\langle p_3 \rangle$ firing t_2. (1) In both cases there is no option to complete, so $z_6 \notin \mathcal{W}_{opt}$. (2) No reachable halting state covers the final one $qe(z_6) = \langle p_4 \rangle$, hence there cannot be an improper completion and $z_6 \in \mathcal{W}_{pro}$. (3) The transition t_3 is dead because it does not appear in any run; it follows $z_6 \notin \mathcal{W}_{nod}$.

z_7 The workflow net $z_7 \in \mathcal{W}_{basic}$ in Fig. 4.5.h, has only one possible run from the initial state $qs(z_7) = \langle p_1 \rangle$, namely $\langle p_1 \rangle \xrightarrow{t_1} \langle p_2 + p_3 \rangle \xrightarrow{t_2} \langle p_3 + p_4 \rangle$. (1) Every reachable state from the initial one can reach $\langle p_3 + p_4 \rangle$ that covers $qe(z_7) = \langle p_4 \rangle$, it follows $z_7 \in \mathcal{W}_{opt}$. (2) There is an halting state $\langle p_3 + p_4 \rangle$ that is not exactly $qe(z_7)$, hence it is an improper completion, $z_7 \notin \mathcal{W}_{pro}$. (3) The transition t_3 is dead because it does not occur in the only possible run, hence $z_7 \notin \mathcal{W}_{nod}$.

z_8 The workflow net $z_8 \in \mathcal{W}_{basic}$ in Fig. 4.5.b, from the initial state $qs(z_8) = \langle p_1 \rangle$, has two possible evolutions, $\langle p_1 \rangle \xrightarrow{t_1} \langle p_2 \rangle$ and $\langle p_1 \rangle \xrightarrow{t_2} \langle p_3 + p_4 \rangle \xrightarrow{t_3} \langle p_3 + p_5 \rangle$. (1) If t_1 is selected then there is no option to complete, $z_8 \notin \mathcal{W}_{opt}$. (2) When t_2 is chosen instead of t_1, there is an improper completion, so $z_8 \notin \mathcal{W}_{pro}$. (3) Clearly, t_4 is a dead transition because it never occurs in one of two possible runs, hence $z_8 \notin \mathcal{W}_{nod}$.

The position of each workflow net is summarized in Fig. 4.6. Finally, it is easy to see that the given soundness requirements are orthogonal, i.e. there is no combination of two of them that can imply the missing one. Formally, assuming $\langle W_i \rangle_{i=1}^{3} = \langle \mathcal{W}_{opt}, \mathcal{W}_{pro}, \mathcal{W}_{nod} \rangle$, one can prove for all $i \in [1,3]$ that $\forall a \in \mathcal{W}_{basic} \,.\, a \in W_j \wedge a \in W_k \Rightarrow a \in W_i$ does not hold, where $j = ((i+1) \mod 3)$ and $k = ((i+2) \mod 3)$. This can be done by finding for any $1 \leq i \leq 3$ a workflow net $a_i \in \mathcal{W}_{basic}$



Net	Properties			Figure
	$\mathcal{W}_{opt}$	$\mathcal{W}_{pro}$	$\mathcal{W}_{nod}$	
z_1	✓	✓	✓	Fig. 4.5.a
z_2	✗	✓	✓	Fig. 4.5.c
z_3	✓	✗	✓	Fig. 4.5.d
z_4	✓	✓	✗	Fig. 4.5.e
z_5	✗	✗	✓	Fig. 4.5.f
z_6	✗	✓	✗	Fig. 4.5.g
z_7	✓	✗	✗	Fig. 4.5.h
z_8	✗	✗	✗	Fig. 4.5.b

Fig. 4.6. A summary of the discussed workflow nets. Any soundness requirement or combination of requirements contains at least one of them.

such that $a_i \in W_j \cap W_k$ but $a_i \notin W_i$. The set $Z = \{z_l\}_{l=1}^{8}$ previously presented contains the required nets, namely $\forall i \in [1,3] \,.\, a_i = z_{i+1}$. □

Proposition 4.12 (Original Soundness is Redundant). The requirements stated in the original notion of soundness [75] are not orthogonal. The original notion of soundness differs from the one given in Def. 4.10 by the "option to complete" requirement that was originally defined as follows:

$$\begin{aligned} &\forall a \in \mathcal{W} \,.\, a \in \mathcal{W}'_{opt} \overset{\triangle}{\iff} \\ &\quad \forall q \in \rho(a, qs(a)) \,.\, \exists r \in \rho(a,q) \,.\, qe(a) \in \rho(a,r) \end{aligned} \tag{4.19}$$

Assuming this original requirement, it can be proven that option to complete implies proper completion [76], formally

$$\forall a \in \mathcal{W} \,.\, a \in \mathcal{W}'_{opt} \Rightarrow a \in \mathcal{W}_{pro} \tag{4.20}$$

Proof. For contradiction, $\exists a \in \mathcal{W}$ such that $a \in \mathcal{W}'_{opt} \wedge a \notin \mathcal{W}_{pro}$.

$$\begin{aligned} a \in \mathcal{W}'_{opt} &\Rightarrow \forall q \in \rho(a, qs(a)) \,.\, \exists r \in \rho(a,q) \,.\, qe(a) \in \rho(a,r) \\ a \notin \mathcal{W}_{pro} &\Rightarrow \exists s \in \rho(a, qs(a)) \,.\, s \geq qe(a) \wedge s \neq qe(a) \end{aligned}$$

if $s \geq qe(a)$ then s is an halt state, it follows $\rho(a,s) = \{s\}$. Applying the first requirement, $\exists r \in \rho(a,s)$ such that $qe(a) \in \rho(a,r)$. But $r \in \rho(a,s) \Rightarrow r = s$, hence $qe(a) \in \rho(a,s)$ that implies $s = qe(a)$, that is a contradiction because $s \neq qe(a)$. □

There is also another caveat regarding the last soundness requirement about dead transitions: in the original definition, a transition is not dead if it can fire

$$\begin{aligned} &\forall a \in \mathcal{W} \,.\, a \in \mathcal{W}'_{nod} \overset{\triangle}{\iff} \\ &\quad \forall t \in trs(a) \,.\, \exists q, r \in \rho(a, qs(a)) \,.\, q \xrightarrow{t} r \end{aligned} \tag{4.21}$$

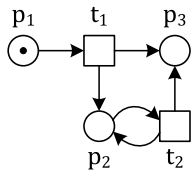


Fig. 4.7. A model that matches the workflow nets definition given in [76].

The formal definition given in Eq. 4.17 is equivalent to this one because no transition can be enabled when there is at least one token in the final place, due to the halting semantics of WFNs. The no dead transitions requirement in [76] is formulated in the same way of Eq. 4.21, but the underlying workflow nets semantics is not clear about termination, making the soundness definition tricky. For instance, the Petri net in Fig. 4.7 is a workflow net w.r.t. the definition given in [76]. If the net does not terminate when a token reaches the final place p_3, then it has an unlimited number of improper completions; hence, the net completes and runs at the same time. If the halting semantics of the original workflow nets [75] is assumed, then the net terminates after firing t_1 with an improper completion. In such case t_2 never fires, but for the soundness definition given in [76], it is also not dead because the final state enables it.

◇

4.5 Soundness Check

Many Petri nets verification methods are based on the construction of the so called reachability graph that can be considered a data structure representing the run-time behavior of a net.

In Lis. 4.1 is reported a basic procedure to construct the reachability graph of a PTNs model $a \in \mathcal{N}$ starting from a valid initial state $q_s \in \mathit{markings}(a)$. The procedure explores all the states reachable from q_s unless the number of such states is greater than the given *max-states* limit. When it terminates, the procedure returns a pair (V, E) representing a directed edge-labeled multi-graph such that $V \subseteq \mathit{markings}(a)$ is its set of vertices and $E \subseteq V \times T \times V$ is its set of edges, where for brevity $T = \mathit{trs}(a)$. The returned reachability graph is complete only if $|V| < \mathit{max\text{-}states}$, otherwise it shall be considered partial.

Listing 4.1 Basic procedure to build the reachability graph of a PTNs model.

input: A net $a \in \mathcal{N}$ expressed in PTNs.
input: A valid initial state $q_s \in \mathit{markings}(a)$.
input: The maximum number of visitable states *max-states*.
output: The complete or partial reachability graph (V, E) of a, represented as a directed edge-labeled multi-graph such that $V \subseteq \mathit{markings}(a)$ and $E \subseteq V \times \mathit{trs}(a) \times V$.

$(V, E) \leftarrow$ BUILD-REACHABILITY-GRAPH$(a, q_s, \mathit{max\text{-}states})$

```
 1  states ← {q_s}
 2  unvisited ← {q_s}
 3  edges ← ∅
 4  while unvisited ≠ ∅ ∧ |states| < max-states do
        // This sub-procedure determines the order in which states are explored.
 5      q ← SELECT-UNVISITED-STATE(unvisited)
 6      unvisited ← unvisited ∖ {q}
 7      for each t ∈ enabled-set(a, q) do
 8          r ← fire(a, q, t)
 9          edges ← edges ∪ {(q, t, r)}
            // If the current state has not been visited before.
10          if r ∉ states then
11              states ← states ∪ {r}
12              unvisited ← unvisited ∪ {r}
13  return (states, edges)
```

The procedure SELECT-UNVISITED-STATE in line 5 of Lis. 4.1 is left unspecified. Such procedure determines the order in which the state space is explored: for instance, a queue data structure can be used for a breath-first search, while a stack can be adopted to obtain a depth-first search. In general, any visiting order that takes care of all unvisited states does not alter the algorithm correctness.

A limit on the maximum number of analyzable states becomes necessary because unboundedness makes the reachability graph not finite. Moreover, even when it is finite, it may be too large to fit in the available memory: a well-known prob-

lem called state-space explosion. For a certain class of Petri nets, the Karp-Miller acceleration [89] can be used to detect unbounded places and to build a so called *coverability graph* that is a finite representation of a potentially unlimited reachability graph. Unfortunately, Karp-Miller acceleration cannot be applied in the analysis of general nets containing special arcs because they are not monotone.

4.5.1 Error Detection

To be useful, a verification method does not need to be complete, i.e. able to detect all errors for which it is thought in any possible model. If a method can detect one or more errors in reasonable time with the available resources, then it can help the end-user during the design activity. Moreover, if the adopted modeling language has some constructs to decompose large models in smaller ones, it is unlikely that the verified models are extremely complex. From this point of view, unboundedness and state-space explosion can become a secondary concern.

Other issues can be related to the abstraction process performed before the verification: excluding the case in which models are designed with the same language used for verification, a mapping becomes necessary. Such mapping usually abstracts from details not considered relevant; hence, the analysis is performed on a simplified model. For example, a mapping can be defined between YAWL and WFNs in order to apply the notion of soundness previously defined. This mapping will abstract from data aspects and concentrate only on control-flow relations. This mapping in turn affects how the detected errors are interpreted in the original model, e.g. an error in the abstract model should become a warning for the original one because its presence cannot be guaranteed. Considering the mapping from YAWL to WFNs, some identified erroneous traces cannot occur in the original model due to the presence of some conditions specified on variables.

From the notion of soundness given in Def. 4.10 four kind of errors can be extracted, each of these is related to one of the three soundness requirements given in Def. 4.7, Def. 4.8 and Def. 4.9. An error of a workflow net $a \in \mathcal{W}$ is defined as a pair $(e, \sigma) \in \mathbb{E} \times \mathcal{T}_R$ such that $e \in \mathbb{E}$ is an error type, where $\mathbb{E} = \{\mathsf{NO}, \mathsf{IC}, \mathsf{DT}, \mathsf{QE}\}$ contains an element for each kind of error, and $\sigma \in \mathcal{T}_R$ is a finite trace of transitions proving the erroneous run. The following definitions are used to characterize each error in a formal way.

Definition 4.13 (No Option to Complete). Let $a \in \mathcal{W}^\bullet$ a marked workflow net with initial marking $qs(a)$, a *no option to complete* error is a pair $(\mathsf{NO}, \sigma) \in \mathbb{E} \times \mathcal{T}_R$ where $\sigma \in \mathcal{T}_R(a)$ is finite sequence of transitions that from $qs(a)$ leads to a terminal marking not covering the final one $qe(a)$. Notice that if exists $q \in \mathcal{Q}$ such that $qs(a) \xrightarrow{\sigma} q$, $q \not\geq qe(a)$ and $\mathit{enabled\text{-}set}(a, q) = \varnothing$ then $\rho(a, q) = \{q\}$ and $q \in \rho(a, qs(a))$, hence $\exists q \in \rho(a, qs(a)) \,.\, \forall r \in \rho(a, q) \,.\, r \not\geq qe(a)$ that implies $a \notin \mathcal{W}_{opt}$. To detect this kind of error it is sufficient to search terminal markings that do not contain tokens in the end place. □

Definition 4.14 (Improper Completion). Let $a \in \mathcal{W}^\bullet$ a marked workflow net with initial marking $qs(a)$, an *improper completion* error is a pair $(\mathsf{IC}, \sigma) \in \mathbb{E} \times \mathcal{T}_R$ where $\sigma \in \mathcal{T}_R(a)$ is a finite sequence of transitions that from $qs(a)$ leads to an halting state different from the final one $qe(a)$. Notice that if exists $q \in \mathcal{Q}$ such

that $qs(a) \xrightarrow{\sigma} q$ and $q > qe(a)$ then $\exists q \in \rho(a, qs(a)) \,.\, q \geq qe(a) \wedge q \neq qe(a)$ that means $a \notin \mathcal{W}_{pro}$. This kind of error can be detected by searching terminal markings that are strictly greater than the final one $qe(a)$. □

Definition 4.15 (Dead Transition). Let $a \in \mathcal{W}^{\bullet}$ a marked workflow net with initial marking $qs(a)$, a *dead transition* error is a pair $(\mathtt{DT}, \sigma) \in \mathbb{E} \times \mathcal{T}_R$ where $\sigma = \langle t \rangle$ represents a single transition $t \in trs(a)$ that never occur in any possible sequence of firings, namely $t \in trs(a)$ is a dead transition if for all reachable states $q \in \rho(a, qs(a))$ it holds that $t \notin enabled\text{-}set(a, q)$. To detect this kind of error the entire reachability graph shall be explored. □

Unfortunately, the construction of the entire reachability graph is not always feasible and a partial one cannot prove that a transition is definitely dead. In face of such limit, it can be useful to define a further error capturing *potential* dead transitions, namely transitions that are never fully enabled during the check. This is reasonable as long as the reachability graph of a workflow net is built using a strategy that aims to enable all transitions, e.g. whenever it can choose between firing an already encountered transition or a new one, the latter is selected.

Definition 4.16 (Ready Places). Given a workflow net $a \in \mathcal{W}$ in a certain state $q \in markings(a)$, the *ready places* of a transition $t \in trs(a)$ are all those places $p \in pre\text{-}set(a, t)$ belonging to its pre-set that meet the firing condition related to the arc $(p, t) \in edges(a)$. The set of ready places is denoted by the function $ready : \mathcal{W} \times \mathcal{Q} \times \mathcal{U} \rightharpoonup \mathcal{U}$ defined as follows:

$$\begin{aligned} &ready : \mathcal{W} \times \mathcal{Q} \times \mathcal{U} \rightharpoonup \mathcal{U} \ \textit{is} \\ &\quad \forall a \in \mathcal{W} \,.\, \forall q \in markings(a) \,.\, \forall t \in trs(a) \,. \\ &\qquad ready(a, q, t) = \{\, p \in pre\text{-}set(a, t) \mid q(p) \geq weight(a, p, t, q) \,\} \end{aligned} \tag{4.22}$$

It is clear from the definition that $ready(a, q, t) \subseteq pre\text{-}set(a, t)$ for every workflow net, valid state and transition. The set of *unready places* is simply given by the pre-set places minus the ready ones, namely $pre\text{-}set(a, t) \setminus ready(a, q, t)$. □

Definition 4.17 (d-Quasi-Enabled Transitions). Let $a \in \mathcal{W}$ be a workflow net in a certain state $q \in markings(a)$ and let $d \in \mathbb{N}$ be the maximum number of admissible unready places. The set of d-quasi-enabled transitions in q is denoted by $qeset : \mathcal{W} \times \mathcal{Q} \times \mathbb{N} \rightharpoonup \mathcal{U}$ and defined in Eq. 4.23, where $n = |pre\text{-}set(a, t)|$ is the size of the pre-set and $r = |ready(a, q, t)|$ the number of ready places.

$$\begin{aligned} &qeset : \mathcal{W} \times \mathcal{Q} \times \mathbb{N} \rightharpoonup \mathcal{U} \ \textit{is} \\ &\quad \forall a \in \mathcal{W} \,.\, \forall q \in markings(a) \,.\, \forall d \in \mathbb{N} \,.\, \forall t \in trs(a) \,. \\ &\qquad t \in qeset(a, q, d) \overset{\triangle}{\Longleftrightarrow} \\ &\qquad\quad (max\{1, n - d\} \leq r \leq n - 1) \ \wedge \ (1 \leq n - r \leq min\,\{d, r - 1\}) \end{aligned} \tag{4.23}$$

For every $a \in \mathcal{W}$, $q \in markings(a)$ and $d \in \mathbb{N}$ it holds that $qeset(a, q, d) \subseteq qeset(a, q, d+1)$, and $qeset(a, q, d) \cap enabled\text{-}set(a, q) = \varnothing$. The number of admissible unready places d becomes global a parameter usually set to 1. □

Definition 4.18 (Quasi-Enabled Transition). Let $a \in \mathcal{W}^{\bullet}$ a marked workflow net that starts its execution from $qs(a)$, a *d-quasi-enabled transition* error is a pair $(\textsf{QE}, \sigma) \in \mathbb{E} \times \mathcal{T}_R$ such that $\sigma = \eta \circ t$, $\eta \in traces(a)$ and $t \in qeset(net(a), q, d)$ where $t \notin enabled\text{-}set(a, r)$ for every state $r \in \mathcal{Q}$ of the reachability graph and $q \in markings(a)$ is the state reachable from $qs(a)$ firing η, namely $qs(a) \xrightarrow{\eta} q$. □

A quasi-enabled transition error is represented by a trace that is near to fulfil the firing condition of a transition $t \in trs(a)$, but without completely satisfying it due to some unready places, provided that the reachability graph constructed so far does not contain t.

Example 4.19 (Workflow Net Errors). Let us consider again the nets in Prop. 4.11 and reported in Fig. 4.8 for convenience. The set of errors characterizing each net are reported below, where the quasi-enabled transitions parameter is set to 1. The net z_1 does not contain errors; z_2 contains $(\textsf{NO}, t_1t_3)$; z_3 contains $(\textsf{IC}, t_1t_2)$ and $(\textsf{IC}, t_1t_3)$; z_4 contains $(\textsf{DT}, t_3)$ and $(\textsf{QE}, t_1t_3)$; z_5 contains $(\textsf{NO}, t_2)$ and $(\textsf{IC}, t_1t_3t_4)$; z_6 contains $(\textsf{NO}, t_1)$, $(\textsf{NO}, t_2)$, $(\textsf{DT}, t_3)$, $(\textsf{QE}, t_1t_3)$ and $(\textsf{QE}, t_1t_2)$; z_7 contains $(\textsf{IC}, t_1t_2)$, $(\textsf{DT}, t_3)$ and $(\textsf{QE}, t_3)$; z_8 contains $(\textsf{IC}, t_1t_2t_3)$, $(\textsf{NO}, t_1)$, $(\textsf{DT}, t_4)$, $(\textsf{QE}, t_1t_4)$ and $(\textsf{QE}, t_1t_2)$. □

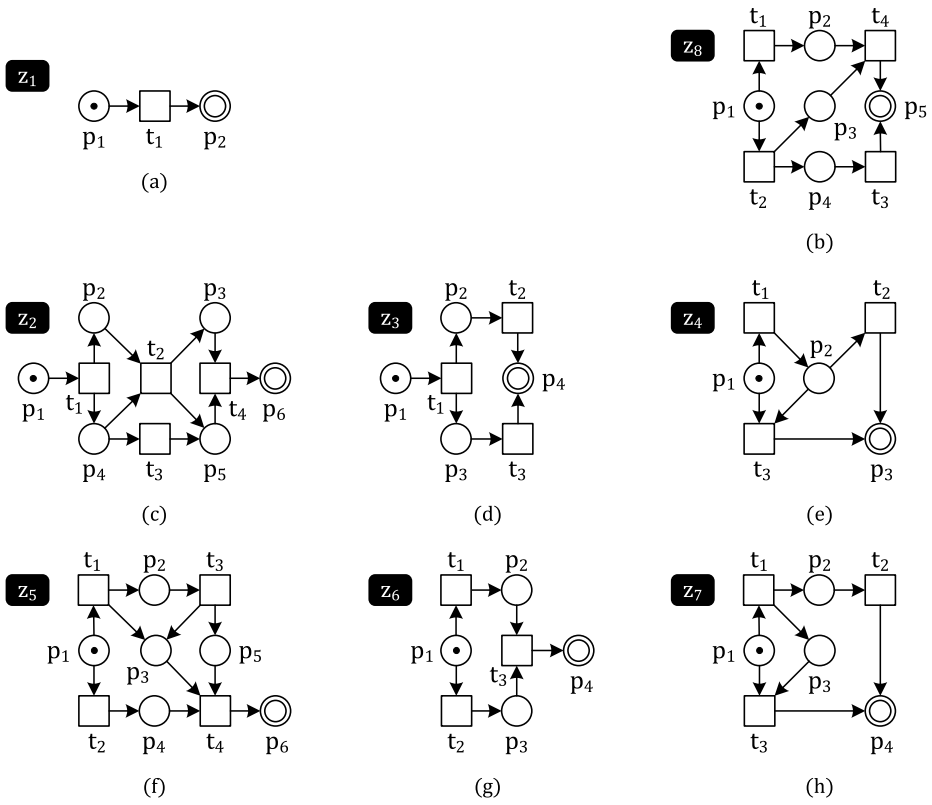


Fig. 4.8. Ordinary workflow nets with different kind of errors.

4.5.2 Soundness Check Algorithm

The soundness check algorithm receives in input a workflow net $a \in \mathcal{W}$, a valid state $q_s \in \mathit{markings}(a)$, a maximum number *max-states* of analyzable states, a number $d \in \mathbb{N}$ of admissible unready places for quasi-enabled transitions, and produces in output the complete or partial reachability graph (V, E) for the net a, together with a finite set R of detected errors. Lis. 4.2 reports the main steps of the algorithm that can be summarized as follows:

1-4 Initialization of the principal variables used by the algorithm. The data structures *states* and *edges* store the vertices and edges of the reachability graph under construction. Each found state that needs to be visited, together with the last transitions for obtaining it, are temporarily stored in the *unvisited* data structure. Such data structure is managed as a queue with priorities. Finally, *errors* is the data structure containing the errors found so far, as pairs composed of a type and a representing trace.

5 *freq* is a mapping $\mathit{freq} : \mathit{trs}(a) \to \mathbb{N} \cup \{0\}$ that tracks the number of times a transition is fired. Initially, the frequency of all transitions is zero, but each frequency increases during the algorithm execution. Given two transitions $t_i, t_j \in \mathit{trs}(a)$, if $\mathit{freq}(t_i) < \mathit{freq}(t_j)$ then t_i is said to be an higher priority than t_j.

6-31 The main loop of the procedure. The loop terminates when all reachable states have been visited or eventually when the *max-states* threshold has been reached.

8-9 Select an unvisited state found during the construction of the reachability graph. The selection takes into account the number of times a transition is fired through the firing frequencies.

10-11 Check if in the current state $q \in \mathit{markings}(a)$ there is no option to complete, namely if $q \neq \mathit{qe}(a)$ and no other transition can fire.

11-13 Check if in the current state $q \in \mathit{markings}(a)$ there is an improper completion, namely if $q > \mathit{qe}(a)$.

15-30 Process each transition $t \in \mathit{trs}(a)$ in order to add all states reachable from the current one to the reachability graph, and to update the set of errors related to quasi-enabled transitions. In particular, for each $t \in \mathit{trs}(a)$ if t is enabled in the current state q, then it is fired producing a new state r. The reachability graph is accordingly updated, the pair (t, r) is added to the *unvisited* queue if necessary, and the frequency of t is incremented. Moreover, on the basis that frequency some errors of quasi-enabled transition for t can be removed. Conversely, for each transition t that is only quasi-enabled in q, the corresponding error is added to *errors*.

32-36 If the analyzable states are exhausted, the reachability graph is complete and any transition that has been never fired is certainly dead.

37 Return to the caller the partial or complete reachability graph and the set containing the found errors.

Listing 4.2 The soundness check procedure.

input: A workflow net $a \in \mathcal{W}$ expressed in WFNs.
input: A valid initial state $q_s \in \mathit{markings}(a)$.
input: The maximum number of analyzable states, *max-states*.
input: The number $d \in \mathbb{N}$ of admissible unready places in a quasi-enabled transition.
output: The complete or partial reachability graph (V, E) of a, represented as a directed edge-labeled multi-graph such that $V \subseteq \mathit{markings}(a)$ and $E \subseteq V \times \mathit{trs}(a) \times V$.
output: A finite set R of errors detected in a.

$(V, E, R) \leftarrow$ Refined-Soundness-Check$(a, q_s, \mathit{max\text{-}states}, d)$

```
1  states ← {q_s}
2  edges ← ∅
3  unvisited ← {(∅, q_s)}
4  errors ← ∅
5  ∀t ∈ trs(a) . freq(t) ← 0
6  while unvisited ≠ ∅ ∧ |states| ≤ max-states do
7      (t, q) ← Select-Unvisited-State(states, edges, unvisited, freq)
8      σ ← Firing-Sequence(states, edges, q)
9      unvisited ← unvisited ∖ {(t, q)}
10     if q ⋡ qe(a) ∧ enabled-set(a, q) = ∅ then
11         errors ← errors ∪ {(NO, σ)}
12     else if q > qe(a) then
13         errors ← errors ∪ {(IC, σ)}
14     end if
15     for each h ∈ trs(a) do
16         if h ∈ enabled-set(a, q) then
17             r ← fire(a, q, h)
18             if r ∉ states then
19                 states ← states ∪ {r}
20                 unvisited ← unvisited ∪ {(h, r)}
21             end if
22             edges ← edges ∪ {(q, h, r)}
23             freq(h) ← freq(h) + 1
24             if freq(h) = 1 then
25                 errors ← errors ∖ {(QE, η) | ∀γ ∈ traces(a) . η = γ ∘ h}
26             end if
27         else if freq(h) = 0 ∧ h ∈ qeset(a, q, d) then
28             errors ← errors ∪ {(QE, σ ∘ h)}
29         end if
30     end for
31 end while
32 if unvisited = ∅ then
33     for each t ∈ trs(a) . freq(t) = 0 do
34         errors ← errors ∪ {(DT, t)}
35     end for
36 end if
37 return (states, edges, errors)
```

The procedure related to the selection of an unvisited state is reported in Lis. 4.3. It receives in input a partial reachability graph, a set of unvisited states, and a mapping computing the priority of each transition. It returns in output a pair (t, q) where q is one of the given unvisited states with maximum priority and t is the last transition performed to produce t. The priority of a state is determined by the frequency of this transition.

1-2 Initialization of the main variables used by the algorithm. The array *list* will contain for each unvisited state (t, q), the tuple $(freq(t), i, q)$ where $freq(t)$ is the frequency of the transition t, i is a sequential number, and q is the state. Variable i tracks the order in which unvisited states are processed.

4-6 The main loop of the procedure. For each unvisited state (t, q) given in input, the frequency of the transition t is evaluated and the tuple $(freq(t), i, q)$ is added to the array *list*.

7-8 The array *list* is sorted with respect to the frequency of each transition in increasing order. In presence of two states with associated the same frequency, the corresponding sequential value i is considered. At the end $list[1]$ contains the unvisited state with highest priority, which is returned to the caller.

Listing 4.3 Selection of an unvisited state.

input: A partial reachability graph (V, E).
input: $U \subseteq V$ the set of unvisited states.
input: $freq : T \to \mathbb{N} \cup \{0\}$ transition priorities, 0 means max priority.
output: An unvisited state $q \in U$ of the reachability graph.

$q \leftarrow$ SELECT-UNVISITED-STATE$(V, E, U, freq)$

```
1  list[ ] array of size |U|
2  i ← 1
3  for each (t, q) ∈ U do
4      list[i] ← (freq(t), i, q)
5      i ← i + 1
6  end for
   // Sort the tuples in list by the frequency of each transition in increasing order.
   // On equal frequency, sort the tuples by the position in U queue.
7  list ← SORT(list)
8  return (t, q) of list[1]
```

◊

4.6 Automated Model Repair

When they are available, simulation and verification tools are routinely used during the design activity to gain confidence about correctness and characteristics of the models at hand. Very often, it is up to the end-user to understand the output produced by these tools and figure out how to fix the detected errors respecting the imposed language constraints. The proposed fix can be seen as an hypothesis that need to be proven by re-running the tools on the rectified instance.

A way to streamline this process is providing some *hints* or *suggestions* to the end-user about how to fix certain errors. As one can imagine, error correction can be a very challenging problem because there are both theoretical and technological limits to consider and there is no way to accurately guess the real design goals: at the end, the evaluation of the proposed hints remains an end-user responsibility.

The problem of finding helpful hints to fix a model is called here *Automated Model Repair* (AMR) problem and can be informally stated as follows: given a model and the data collected during the verification phase, search a small set of models, called here *solutions*, that are similar to the original one but contain fewer errors, such that no solution can be considered worse than any other. More than one solution is necessary because the end-user's intentions are not known a priori and there are usually many ways in which an error can be corrected. The restriction that no solution should be worse than any other is imposed to reduce redundancy in the proposed fixes. The structural differences between these solutions and the original model can then be presented as hints to fix the related errors.

In the remaining part of this section the AMR problem is restated in more formal terms for models expressed in the WFNs language introduced in Sec. 4.3. The choice of WFNs is driven by these two considerations: firstly, the AMR problem has no established solutions out of the box, hence it is reasonable to initially study it on simpler languages; secondly, as regularly happens for existing verification techniques, an AMR solution for WFNs can be applied to real PMLs by mapping each model under analysis to a workflow net abstracting away undesired details. The problem of mapping real PMLs to Petri nets is not considered any further because several solutions already exist, see for instance the work in [90].

Loosely speaking, a Petri net that does not exhibit the formal requirements stated in Def. 4.3 can be still considered a workflow net containing one or more syntactical errors. This definition becomes useful in describing a design activity, because not all operations performed by the end-user produce syntactically correct nets, especially if the adopted editor does not enforce them. In the following, a workflow net that complies with its formal definition is remarked to be *valid*.

Checking and enforcing syntactical constraints is not a big challenge, hence they will not be explicitly considered as errors to be repaired, provided that any AMR solution produces valid models. AMR explicitly focuses on semantical requirements, i.e. restrictions on how a process model shall behave at run-time. In the context of WFNs the considered errors are those derived by the notion of soundness extensively discussed in Sec. 4.5: no option to complete, improper completion, dead transitions and quasi-enabled ones. The following Ex. 4.20 shows what can happen when a workflow net is edited for adding a new final transition.

Example 4.20. In Fig. 4.9.a is depicted a workflow net $a_1 \in \mathcal{W}_{basic}$ with seven places $pls(a_1) = \{p_i\}_{i=1}^{7}$, six transitions $trs(a_1) = \{t_j\}_{j=1}^{6}$ and a simple loop $\{p_4, t_4, p_5, t_5, p_6, t_6\}$. The model has a single initial place $start(a_1) = p_1$, a single end place $end(a_1) = p_7$ and any transition lies in a path from p_1 to p_7, hence it is also a valid workflow net. The model in Fig. 4.9.b is another valid workflow net $a_2 \in \mathcal{W}_{basic}$ obtained from a_1 by adding two places p_8 and p_9, a new transition t_7 and the missing edges. The added transition t_7 cannot fire before at least one execution of t_3 and t_5. The net a_1 in Fig. 4.9.a is sound with respect to the definition given Sec. 4.4: any state reachable from the initial one can reach the final state, when a token is placed in p_7 no other token is left in the net and also no transition can fire. The net a_2 in Fig. 4.9.b is obtained from a_1 with a small change but is no longer sound: a no option to complete is reached when t_3 fires immediately after the first fire of t_4, because the terminal state $p_7 \not\geq p_9$ is reached, and an improper completion is reached whenever the loop $\{p_4, t_4, p_5, t_5, p_6, t_6\}$ is run more than one time, because the state $p_7 + n \cdot p_8$ is reached where $n + 1$ is the number of performed loops. □

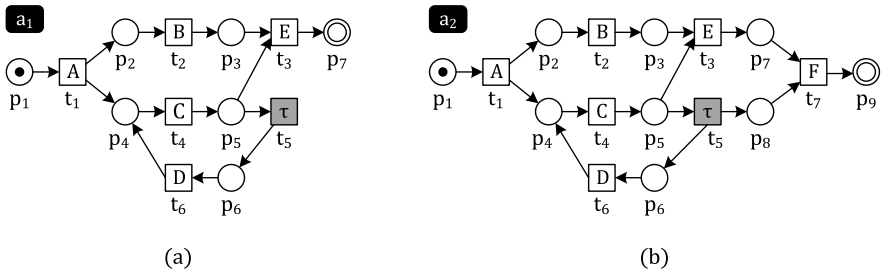


Fig. 4.9. (a) A workflow net with a simple loop $\{p_4, t_4, p_5, t_5, p_6, t_6\}$. (b) A new version of the net obtained by adding a final transition t_7 that fires after t_3 and t_5.

The similarity requirement stated for the AMR problem deserves a brief discussion: a model can be considered a useful solution only if it is similar to the original one. Obviously, a solution that improves an unsound workflow net shall preserve as much as possible the observable behavior of the original one: otherwise, the AMR problem becomes trivial. If behavioral similarity is not required, a workflow net having the same set of transitions of the original one arranged in sequence is an optimal solution. For instance, consider the workflow net $b1 \in \mathcal{W}$ in Fig. 4.10.a: it has a dead transition t_4 but it can be made sound removing the arc (p_2, t_4). The workflow net $b2 \in \mathcal{W}$ in Fig. 4.10.b is sound, it contains the same transitions of $b1$ but can be hardly considered a good solution because it does not resemble at all the original net behavior.

Structural similarity is also essential for both characterizing a useful solution and making an AMR method feasible. From one hand, a good solution with an observable behavior close to the original one but with a very different structure is worthless for the end-user: if the information conveyed by the structure is discarded, it can be hardly recognized as a useful hint. For instance, the workflow net $b3 \in \mathcal{W}$ in Fig. 4.10.c is sound and has exactly the same behavior of $b1$ after

removing the arc (p_2, t_4), hence it could be considered a good solution but it has a really different structure.

From the other hand, as recognized by the basic genetic programming principles, structural similarity reduces the probability of introducing new errors in other parts of the model not contemplated in the analysis. This principle is of utmost importance, especially when it cannot be guaranteed that a solution is completely free from errors, either because the problem is generally undecidable or because it is not feasible for real problem instances.

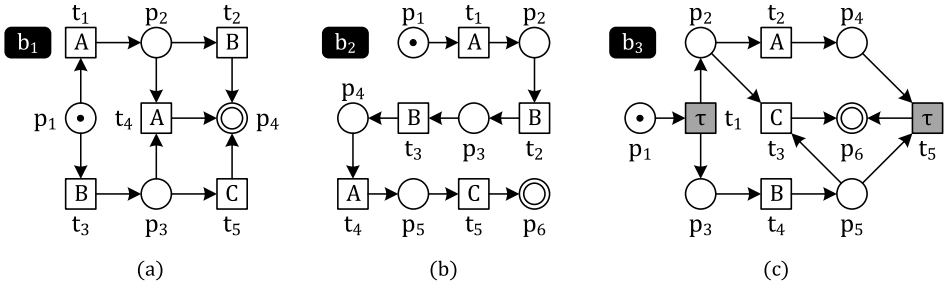


Fig. 4.10. (a) An workflow net b_1 with a dead transition t_4. (b) A sound workflow net b_2 with the same transitions of b_1 but not exposing the same behavior. (c) A sound workflow net b_3 behaviorally similar to b_1 but with a very different structure.

Definition 4.21 (AMR Problem). Given an unsound workflow net $a \in \mathcal{W}$ with a finite non-empty set of errors $errors(a) \neq \emptyset$ and given a finite quantity of computational resources that limits the set of analyzable nets $C \subseteq \mathcal{W}$, find whenever possible a small set of at most $n \in \mathbb{N}$ solutions $S = \{s_i\}_{i=1}^{k} \subseteq C$ with $0 \leq k \leq n$ such that for all $i \in [1, k]$ the following six requirements hold:

1. The solution s_i is structurally similar to the original model a;
2. The solution s_i is behaviorally similar to the original model a;
3. The solution s_i contains less errors than a, i.e. $|errors(s_i)| < |errors(a)|$;
4. The solution s_i is not strictly better than the remaining ones $S \setminus \{s_i\}$;
5. The solution s_i is not worse than any other considered model in $C \setminus S$;
6. Optionally, the solution s_i does not contain new errors with respect to the original model a, i.e. $errors(s_i) \subseteq errors(a)$.

The notion of error mostly depends on the adopted verification methods, while the notions of structural and behavioral similarity are specific of the proposed solution and will be formalized later. The set S of selected solutions may be not unique: there can be more than n equally better solutions in C with respect to the original net a. The parameter n is used to limit the number of solutions proposed to the end-user, while C represents some limits on the available resources, and can be equal to $\mathcal{W}$ if no explicit constraints are imposed, i.e. the AMR procedure runs until the end-user decides to stop it, either because the hints offered so far are good enough or the procedure takes too much time in producing another result.

Example 4.22. For convenience Fig. 4.11.a reports the unsound workflow net a_2 presented in Ex. 4.20. Such net can be fixed in several different ways: three possible solutions are depicted from Fig. 4.11.b to Fig. 4.11.d. The model a_3 in Fig. 4.11.b is obtained removing the arc (p_5, t_3). The model a_4 in Fig. 4.11.c is the result of a place merge between p_6 and p_8 and an according moving of the arc (t_5, p_6). From a different perspective, this operation is equivalent to remove p_6 and its related arcs and subsequently connect p_8 to t_6. The last model a_5 in Fig. 4.11.d presents the same place simplification, but with an additional arc from t_3 to p_5.

The differences between a found solution and the original model can be proposed to the end-user as a hint for fixing a particular error. For instance, considering the solution a_4 in Fig. 4.11.c, a hint for fixing the original model a_2 in Fig. 4.11.a can be to delete p_6 and add (p_8, t_6). □

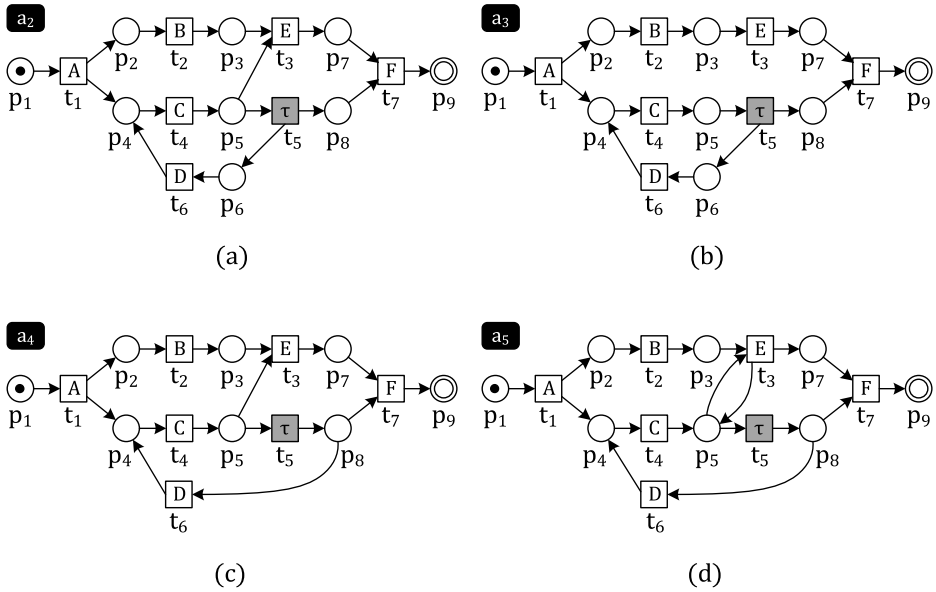


Fig. 4.11. (a) The unsound workflow net a_2 of Fig. 4.9.b in Ex. 4.20. (b-d) three similar workflow nets derived from a_2 to fix the detected errors.

◊

4.7 Petri Nets Simulated Annealing

An exact solution for the AMR problem can be found by restating it as an optimization problem: let $errors(x)$ the set of errors detected in $x \in \mathcal{W}$ by the soundness check, and let $stdist : \mathcal{W} \times \mathcal{W} \to \mathbb{R}$ and $bhdist : \mathcal{W} \times \mathcal{W} \to \mathbb{R}$ two functions measuring the structural and behavioral distance between two workflow nets, respectively. Then, a solution $y \in \mathcal{W}$ can be found by minimizing $stdist(x, y)$, $bhdist(x, y)$ and $|errors(y)|$ subject to the optional constraint that $errors(y) \subseteq errors(x)$, whenever introducing new errors is not acceptable. In particular, considering a workflow net as a graph enriched with some attributes, the $stdist : \mathcal{W} \times \mathcal{W} \to \mathbb{R}$ function can be defined in terms of graph edit distance [91]. In similar way, $bhdist : \mathcal{W} \times \mathcal{W} \to \mathbb{R}$ can be defined in terms of transition adjacency relation (TAR) as done in [92]. The optional constraint about error inclusion can be enforced at least for basic workflow nets running multiple times the soundness check procedure. Unfortunately, these functions have high computational complexities and they need to be evaluated several times for pruning the search space. In particular both the TAR metric adopted as behavioral distance and the soundness check require the exploration of the entire state-space that can take exponential time in the worst case.

This section introduces a novel GP technique called *Petri Nets Simulated Annealing (PNSA)* that aims to solve the AMR optimization problem in a clever way using approximated objective functions. The PNSA technique is essentially a heuristic optimization algorithm with at the core a procedure similar to dominance-based *Multi-Objective Simulated Annealing (MOSA)* [85, 86]: such procedure searches for solutions that minimize structural distance, behavioral distance and the number of detected errors. Due to approximations, there is a chance of considering as good a wrong solution; hence, at the same time the procedure maximizes the confidence about the selected models. The obtained solutions can be offered as they are to the end-user or further refined with an extensive check to guarantee their optimality. The remainder of this section introduces the key ideas behind each objective function and the relevant steps of the PNSA technique.

4.7.1 Structural Similarity

Let $\mathcal{G}_A$ be the set of all attributed graphs $(V, E, \Sigma, \mu, \eta)$ such that (V, E) is a directed graph augmented with two functions $\mu : V \to \Sigma^*$ and $\eta : E \to \Sigma^*$ that assign to each of its components zero or more attributes chosen from a predefined attribute set Σ. The structural distance between two workflow nets $x, y \in \mathcal{W}$ is defined in terms of the graph edit distance [91] between two attributed graphs obtained from the initial nets through the conversion function $\gamma : \mathcal{W} \to \mathcal{G}_A$ defined as follows.

Definition 4.23 (Conversion Function). Given a workflow net $a \in \mathcal{W}$, the conversion function $\gamma : \mathcal{W} \to \mathcal{G}_A$ transforms a into an equivalent attributed graph $\gamma(a) = (V, E, \Sigma, \mu, \eta) \in \mathcal{G}_A$ such that

$$
\begin{aligned}
&\gamma : \mathcal{W} \to \mathcal{G}_A \textit{ is} \\
&\quad \forall a \in \mathcal{W} \,.\, \forall g \in \mathcal{G}_A \,.\, \gamma(a) = g \overset{\Delta}{\iff} \\
&\qquad g = (\textit{vertices}(a), \textit{edges}(a), \Sigma, \mu, \eta) \wedge \\
&\qquad \Sigma = \{\mathsf{PL}, \mathsf{TR}\} \cup \textit{actions}(a) \cup \{\tau\} \cup \{\bot\} \;\wedge \\
&\qquad \forall v \in \textit{vertices}(a) \,.\, \mu(v) = \begin{cases} (\mathsf{TR}, \lambda(a, v)) & \text{if } v \in \textit{trs}(a) \\ (\mathsf{PL}, \bot) & \text{if } v \in \textit{pls}(a) \end{cases} \wedge \\
&\qquad \forall e \in \textit{edges}(a) \,.\, \eta(e) = \varnothing
\end{aligned}
\tag{4.24}
$$

where $\lambda(a, v)$ is defined in Sec. 3.2.3 and assigns to transition v in a its action.

Definition 4.24 (Structural Distance). The structural distance between two workflow nets $x, y \in \mathcal{W}$ is the graph edit distance $ged : \mathcal{G}_A \times \mathcal{G}_A \to \mathbb{R}$ between the corresponding attributed graphs. The notion of structural distance is captured by the function $stdist : \mathcal{W} \times \mathcal{W} \to \mathbb{R}$ defined as follows:

$$
\forall x, y \in \mathcal{W} \,.\, \textit{stdist}(x, y) = \textit{ged}(\gamma(x), \gamma(y)) = \min_{\langle e_i \rangle_{i=1}^{k} \in \Upsilon(x,y)} \sum_{i=1}^{k} c(e_i)
$$

where $\Upsilon(x, y)$ is the set of all edit scripts $\langle e_i \rangle_{i=1}^{k}$ between the graphs $\gamma(x)$ and $\gamma(y)$, and $c : \mathcal{U} \to \mathbb{R}$ is a function that assigns a cost to each edit operation e_i.

The permitted edit operations on graphs generally includes insertion, deletion and substitution of nodes and edges. In PNSA, the cost of a node or an edge substitution is set to be greater than the cost of a deletion followed by an insert operation, hence a substitution is never part of a minimum edit path.

Moreover, while it is safe to add or remove places and arcs, transitions need a special treatment. Since the end-user intentions are not known, it is safer to assume that a visible action has been introduced for a specific purpose, hence its related transition should not be removed from the model. On the other hand, a silent action is mostly used for routing purposes, hence it can be safely removed or inserted without the risk of losing relevant information.

The cost of each type of edit operation can be controlled by the end-user. For example, one may rate removal operations as more expensive than insertion operations, assuming less likely that something erroneous has been introduced than something essential has been forgotten. Similarly, one may rate operations on transitions as more expensive than operations on places, and the latter as more expensive than operations on arcs.

Example 4.25. Coming back to the workflow nets of Ex. 4.22, reported for convenience below in Fig. 4.12. The workflow nets from a_3 to a_5 are all solutions of a_2 which can be obtained via one or more edit sequences. In particular, a_3 can be obtained directly from a_2 with an edit sequence consisting of a single edit operation that removes the arc (p_5, t_3). The solution a_4 can be obtained with four edit operations, the deletion of p_6 and its surrounding arcs followed by the insertion of a new arc (p_8, t_6). The edit sequence of a_5 is the same of a_4 plus a new arc (t_3, p_5).

Following the consideration given above for the cost of each edit operation, we can assign a unitary cost to each arc addition or removal, and a cost equal to one

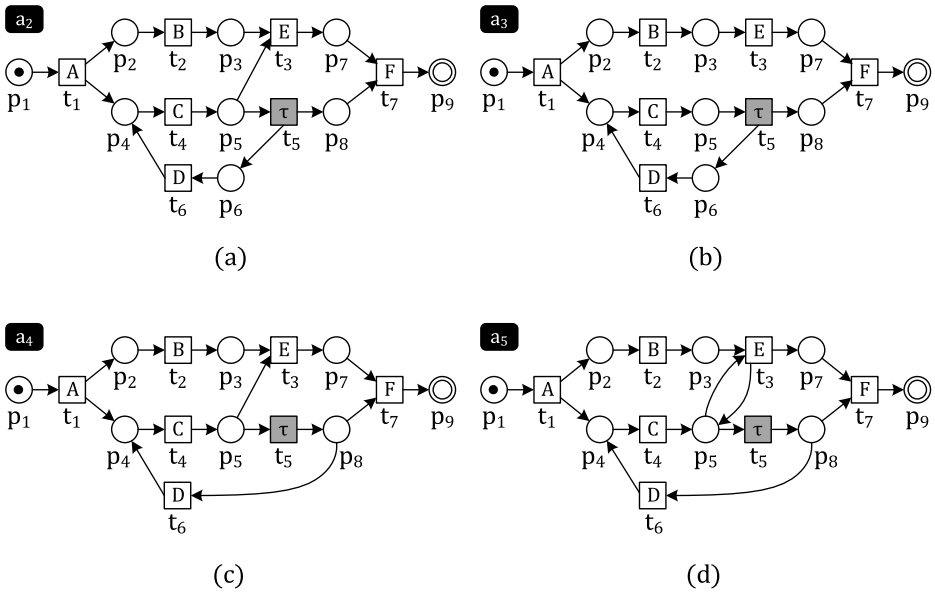


Fig. 4.12. The workflow nets introduced in Ex. 4.22 reported here for convenience.

plus the edit cost for the connected arcs to each place addition or removal. In this case, $stdist(a_2, a_3) = 1$, $stdist(a_2, a_4) = 4$, and $stdist(a_2, a_5) = 5$. □

The graph edit distance between two graphs can be computed using one of the many available algorithms, such as the one proposed in [91]. Computing the graph edit distance can be expensive: fortunately, the presented technique produces a candidate solution $y \in \mathcal{W}$ by performing a sequence of perturbations on the original workflow net $x \in \mathcal{W}$, and these perturbations are essentially a sequence of edit operations. The structural distance between x and y can be therefore computed by minimizing the perturbation sequence.

4.7.2 Behavioral Similarity

The notion of behavioral similarity is more complex than structural similarity because it needs to be approximated in some way. Here, the behavioral distance between two workflow nets $x, y \in \mathcal{W}$ is defined as the difference between their capability to simulate a finite set of traces $R \subseteq \mathcal{T}_R$, where both the simulation capability and the set R are parameters of the distance. To draw an analogy, the behavioral distance between two individuals can be estimated by using a good set of tests representing the expected behavior, under the assumption that both individuals want to participate doing their best for the evaluation.

Simulating a trace of transitions in a workflow net is not a big problem because each transition uniquely identifies the path to take. On the contrary, simulating a trace of visible actions is in general undecidable due to the presence of internal silent actions: in an attempt to enable the wanted visible action, any silent action encountered during the simulation can open a new search path. For this reason, the

focus of simulating a visible trace is not about deciding if that trace is completely recognized by the workflow net, but on how many actions can be performed before halting subject to certain resource limits, e.g. a limit on the number of silent transitions that can be fired during the simulation. For convenience, the notion of behavioral distance is made independent from these details through the concept of trace simulation.

Definition 4.26 (Trace Simulation). A *trace simulation* is a function $h : \mathcal{W} \times \mathcal{T}_R \to \mathcal{T}_R$ that given a workflow net $x \in \mathcal{W}$ and a trace of transitions $\sigma \in \mathcal{T}_R$ returns the prefix of σ that x is able to simulate preserving visible actions. The set of all trace simulations is denoted by $\mathcal{H}$ and defined as follows:

$$\begin{aligned} \forall h \in \mathcal{U} \,.\, h \in \mathcal{H} \mathrel{\triangleq\!\!\Longleftrightarrow} h : \mathcal{W} \times \mathcal{T}_R \to \mathcal{T}_R \wedge \\ \forall x \in \mathcal{W} \,.\, \forall \sigma \in \mathcal{T}_R \,. \\ \sigma \in \mathcal{T}_R(x) \Rightarrow h(x, \sigma) = \sigma \wedge \\ obs(x, h(x, \sigma)) \in prefixes\,(obs(x, \sigma)) \end{aligned} \tag{4.25}$$

where the function $obs : \mathcal{N} \times \mathcal{T}_R \to \mathcal{T}_A$ is defined in Def. 3.15, it returns the corresponding sequence of visible actions of the specified trace of transitions, while the notion of prefixes is explained in Chap. 2.

Example 4.27. Let us reconsider the workflow nets of Ex. 4.22, reported in Fig. 4.13 for convenience, and the trace $\sigma = t_1 t_2 t_4 t_5 t_6 t_4 t_3 t_7$. Clearly, $\sigma \in traces(a_2)$ and $obs(a_2, \sigma) = \langle A, B, C, D, C, E, F \rangle$. A trace simulation that fires at least a silent transition can fully simulate the visible trace on a_3 in Fig. 4.13.b. On the contrary, no trace simulation can completely replay the visible trace on a_4 in Fig. 4.13.c because t_7 is a dead transition. Finally, the visible trace can be completely simulated on a_5 in Fig. 4.13.d assuming that the adopted trace simulation is able to fire at least two silent actions. □

Definition 4.28 (Behavioral Distance). The behavioral distance between two workflow nets $x, y \in \mathcal{W}$ with respect to a trace simulation $h \in \mathcal{H}$ and a representative set of traces $R \subseteq \mathcal{T}_R$, is given by the difference between the length of their best possible simulation of each $\sigma \in R$. The behavioral distance is captured by the following function $bhdist : \mathcal{W} \times \mathcal{W} \times \mathcal{H} \times \wp(\mathcal{T}_R) \to \mathbb{R}$ define as

$$\begin{aligned} bhdist : \mathcal{W} \times \mathcal{W} \times \mathcal{H} \times \wp(\mathcal{T}_R) \to \mathbb{R} \;\; is \\ \forall x, y \in \mathcal{W} \,.\, \forall h \in \mathcal{H} \,.\, \forall R \subseteq \mathcal{T}_R \,. \\ bhdist(x, y, h, R) = \sum_{\sigma \in R} \frac{\big|\, |obs(x, h(x, \sigma))| - |obs(y, h(y, \sigma))| \,\big|}{max\{1, |\sigma|\}} \end{aligned} \tag{4.26}$$

The effectiveness of the behavioral distance $bhdist : \mathcal{W} \times \mathcal{W} \times \mathcal{H} \times \wp(\mathcal{T}_R) \to \mathbb{R}$ in distinguishing two workflow nets depends on the quality of both the chosen parameters $h \in \mathcal{H}$ and $R \subseteq \mathcal{T}_R$. In particular, not all trace simulations are equal: to be effective, behavioural distance shall use a trace simulation that always performs its best by returning the longest possible simulation.

Proposition 4.29. Let $h \in \mathcal{H}$ be a trace simulation and $R \subseteq \mathcal{T}_R$ a predefined set of traces, the defined behavioral distance is a pseudo-metric: $\forall x, y, z \in \mathcal{W}$ it holds

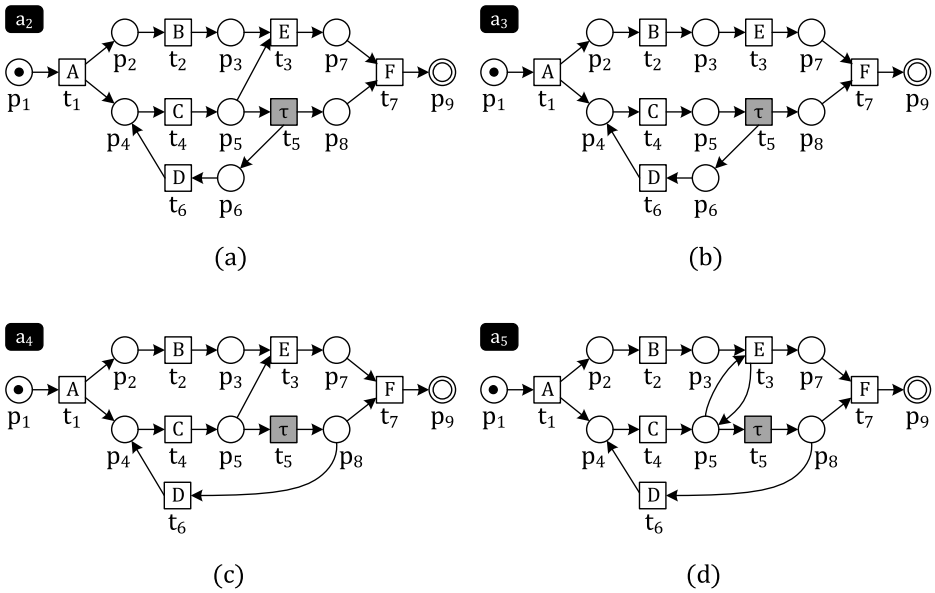


Fig. 4.13. The workflow nets introduced in Ex. 4.22 reported here for convenience.

that (1) $bhdist(x, y, h, R) \geq 0$, (2) $bhdist(x, x, h, R) = 0$, (3) $bhdist(x, y, h, R) = bhdist(y, x, h, R)$, and (4) $bhdist(x, y, h, R) + bhdist(y, z, h, R) \geq bhdist(x, z, h, R)$.

Proof. Property (1) is trivial since function *bhdist* is a sum of non negative values: for each fraction in the sum, the term above is an absolute value and the term below is a trace length which cannot be negative. Relatively to property (2) we can notice that if the two nets are the same, for each trace $\sigma \in R$ its corresponding observable trace has the same length, hence each summation element is zero. Similarly, property (3) is true because for each summation element the numerator is wrapped by an absolute value and the net order can be safely inverted. Finally, as regards to property (4) we can observe that for each summation elements $max\{1, |\sigma|\}$ is the same for any trace σ in R, hence we can discard the denominator and try to prove the relation:

$$\begin{aligned} &| \, |obs(x, h(x, \sigma))| - |obs(y, h(y, \sigma))| \, | + | \, |obs(y, h(y, \sigma))| - |obs(z, h(z, \sigma))| \, | \geq \\ &\qquad | \, |obs(x, h(x, \sigma))| - |obs(z, h(z, \sigma))| \, | \end{aligned} \tag{4.27}$$

Let $|obs(x, h(x, \sigma))| = a$, $|obs(y, h(y, \sigma))| = b$, and $|obs(z, h(z, \sigma))| = c$, Eq. 4.27 becomes: $|a - b| + |b - c| \geq |a - c|$.

1. Suppose $(a - b) \geq 0 \wedge (b - c) \geq 0 \Rightarrow a \geq b \geq c \Rightarrow (a - c) \geq 0$:

$$\begin{aligned} |a - b| + |b - c| &\geq |a - c| \\ a - b + b - c &\geq a - c \\ a - c &\geq a - c \end{aligned}$$

which is trivially true.

2. Suppose $(a-b) \geq 0 \wedge (b-c) < 0 \Rightarrow a \geq b \wedge b < c$:

   - $(a-c) \geq 0 \Rightarrow a \geq c$

   $$\begin{aligned} |a-b| + |b-c| &\geq |a-c| \\ a-b-b+c &\geq a-c \\ -2b+c &\geq -c \\ 2b &< 2c \end{aligned}$$

   which is true for hypothesis $(b-c) < 0$.
   - $(a-c) < 0 \Rightarrow a < c$

   $$\begin{aligned} |a-b| + |b-c| &\geq |a-c| \\ a-b-b+c &\geq c-a \\ a-2b &\geq -a \\ 2b &< 2a \end{aligned}$$

   which is true for hypothesis $(a-b) \geq 0$.
3. Suppose $(a-b) < 0 \wedge (b-c) \geq 0 \Rightarrow a < b \wedge b \geq c$: the proof is similar to case 2.
4. Suppose $(a-b) < 0 \wedge (b-c) < 0 \Rightarrow a < b < c \Rightarrow (a-c) < 0$: the proof is similar to case 1. □

Example 4.30. With reference to the workflow nets of Ex. 4.22, reported in Fig. 4.13, let us consider the set of traces $R = \{t_1t_2t_4t_5t_6t_4t_3t_7, t_1t_2t_4t_5t_6t_4t_5t_6t_4t_3t_7, t_1t_2t_4t_3\}$ where the first two traces are extracted from $traces(a_2)$ and $h \in \mathcal{H}$ a trace simulator. The behavioral distance between a_2 the nets a_3, a_4, and a_5 can be computed as follows:

$$\begin{aligned} & bhdist(a_2, a_3, h, R) = \frac{|7-7|}{8} + \frac{|9-9|}{11} + \frac{|4-4|}{4} = 0 \\ & bhdist(a_2, a_4, h, R) = \frac{|7-6|}{8} + \frac{|9-8|}{11} + \frac{|4-4|}{4} = 0.22 \\ \tau = 2 \quad & bhdist(a_2, a_5, h, R) = \frac{|7-7|}{8} + \frac{|9-9|}{11} + \frac{|4-4|}{4} = 0 \\ \tau = 1 \quad & bhdist(a_2, a_5, h, R) = \frac{|7-6|}{8} + \frac{|9-8|}{11} + \frac{|4-4|}{4} = 0.22 \end{aligned}$$

Notice that for net a_5 different results can be obtained by allowing a total number of silent transitions equal to 2 or to 1. □

◊

4.7.3 Badness

Broadly speaking, a workflow net is better than another one if it contains less errors, but not all errors are equal and errors of the same type may have a different severity. Furthermore, depending on the applied verification methods, certain errors should be interpreted only as warnings because there is a chance of false positives. For instance, the soundness check presented in Sec. 4.5 can ensure that a transition $t \in trs(a)$ of a net $a \in \mathcal{W}$ is definitely dead only if the entire state-space of a has been explored. For this reasons, it becomes useful to score each error on the basis of its severity, rather than rely only on the number of detected errors. The scoring is captured by the notion of *badness* discussed in this section, that in turn depends on two parameters like the behavioral distance, a trace simulation $h \in \mathcal{H}$ and a representative set of traces $R \subseteq \mathcal{T}_R$.

Definition 4.31 (Badness). Let $x \in \mathcal{W}$ a workflow net with initial marking $qs(x)$, the *badness* of x with respect to a set of traces $R \subseteq \mathcal{T}_R$ and a simulation function $h \in \mathcal{H}$ is captured by the function $badness : \mathcal{W} \times \mathcal{H} \times \wp(\mathcal{T}_R) \to \mathbb{R}$ defined below:

$$\begin{aligned} &badness : \mathcal{W} \times \mathcal{H} \times \wp(\mathcal{T}_R) \to \mathbb{R} \ is \\ &\quad \forall x \in \mathcal{W} \,.\, \forall h \in \mathcal{H} \,.\, \forall R \subseteq \mathcal{T}_R \,. \\ &\qquad badness(x, h, R) = \sum_{\sigma \in R} \left(\sum_{i=1}^{3} c_i \cdot \beta_i(x, h(x, \sigma)) \right) \end{aligned} \tag{4.28}$$

Each scoring function $\{\beta_i\}_{i=1}^{3} \subseteq \mathcal{W} \times \mathcal{T}_R \to \mathbb{R}$ is used to score a particular error and the coefficients $\{c_i\}_{i=1}^{3} \subseteq \mathbb{R}$ reflect the relative importance of such errors. The functions $\{\beta_i\}_{i=1}^{3}$ are defined as follows:

$$\beta_1(x, \sigma) = \frac{dvts(\sigma)}{|trs(x)|} \cdot \frac{1 - halt(x, q)}{1 + nocs(\sigma)} \cdot terminal(x, q) \tag{4.29}$$

$$\beta_2(x, \sigma) = \frac{dvts(\sigma)}{|trs(x)|} \cdot \frac{|q \ominus q(end(x))|}{dvps(\sigma)} \cdot halt(x, q) \tag{4.30}$$

$$\beta_3(x, \sigma) = \frac{1}{dvts(\sigma)} \cdot \frac{n - r}{n} \cdot |qeset(x, q, d) \cap \{end(\sigma)\}| \tag{4.31}$$

where $q = \textit{fire-trace}(x, qs(x), \sigma)$ is the final state reachable running the trace σ, $n = |\textit{pre-set}(x, end(\sigma))|$ is the size of the pre-set of the last transition of σ, and $r = |ready(x, q, end(\sigma))|$ the number of ready places of $end(\sigma)$ in q.

The auxiliary functions $dvts : \mathcal{T}_R \to \mathbb{N} \cup \{0\}$ and $dvps : \mathcal{T}_R \to \mathbb{N} \cup \{0\}$ denote respectively the number of distinct transitions and places visited during the execution of σ, and $nocs : \mathcal{T}_R \to \mathbb{N} \cup \{0\}$ is the number of choices contained in the trace σ. The function $terminal : \mathcal{W} \times \mathcal{Q} \to \{0, 1\}$ is defined such that for all $x \in \mathcal{W}$ and $q \in markings(x)$, $terminal(x, q) = 1$ if $\textit{enabled-set}(x, q) = \varnothing$, 0 otherwise. The function $halt : \mathcal{W} \times \mathcal{Q} \to \{0, 1\}$ is defined in a similar way such that $halt(x, q) = 1$ if $q \geq qe(x)$, 0 otherwise, for every $x \in \mathcal{W}$ and $q \in markings(x)$. □

Function $\beta_1(x, \sigma)$ in Eq. 4.29 returns a value different from zero if and only if the trace σ leads to a state q representing a no option to complete error. Indeed,

a no option to complete is characterized by a terminal state that is not a non halt state, if one of these conditions is not satisfied the function returns zero. Conversely, function $\beta_2(x, \sigma)$ in Eq. 4.30 captures the notion of improper completion, because it returns a value different from zero if and only if the trace σ leads to an halt state which covers the end state ($|q \ominus q(end(x))| \neq 0$). Finally, function $\beta_3(x, \sigma)$ in Eq. 4.31 rates errors related to (potentially) dead transitions. Indeed, it returns a value different from zero if and only if the last transition in σ cannot be performed and the number of places in its preset n is less than the number of ready places r.

The coefficients $\{c_i\}_{i=1}^{3}$ determine the relative weight of each scoring function and they should be intended as global parameters; hence, they do not appear as parameters of the badness function. These coefficients can be used to tune the badness computation, e.g. a no option to complete can be considered more serious than an improper completion, because it potentially prevents the execution of parts of the workflow net that are not reachable in other alternative ways.

Example 4.32. Let us reconsider the workflow nets of Ex. 4.22, reported in Fig. 4.14 for convenience, and the set of traces $R = \{t_1t_2t_4t_5t_6t_4t_3t_7,\ t_1t_2t_4t_5t_6t_4t_5t_6t_4t_3t_7,\ t_1t_2t_4t_3\}$. Tab. 4.32 reports for each net the final state reachable running each trace and the type of error eventually generated. Notice that trace σ_3 leads for nets a_3 and a_5 to a state $p_5 + p_7$ which does not represent any error, since the execution can continue.

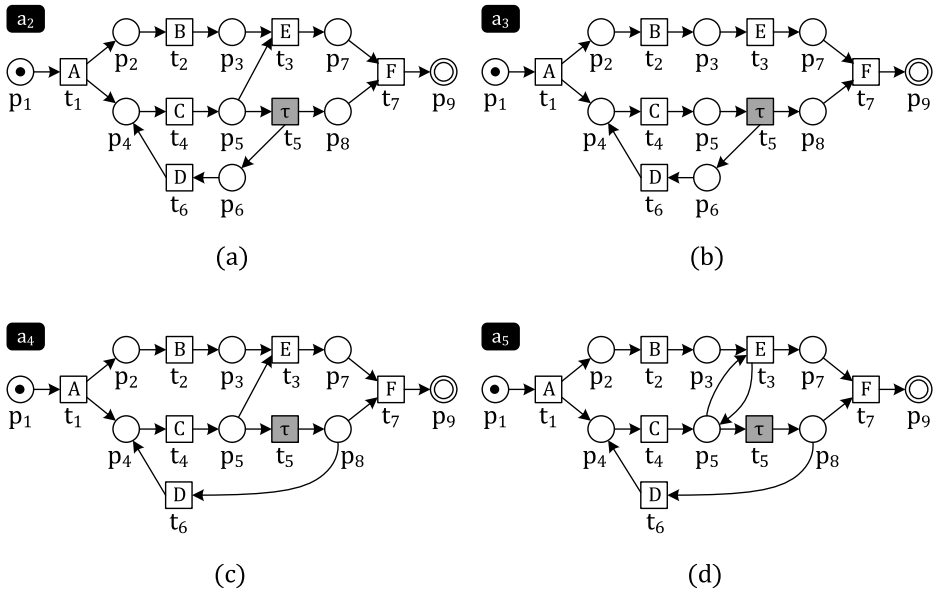


Fig. 4.14. The workflow nets introduced in Ex. 4.22 reported here for convenience.

For each net the badness can be computed as follows:

a_2 Trace σ_1 is a correct trace, it leads to state $q = p_9$ and $badness(\sigma_1) = 0$:

Trace	a_2		a_3		a_4		a_5
$\sigma_1 = t_1t_2t_4t_5t_6t_4t_3t_7$	p_9		$p_5 + p_9$	IC	p_7	QE $[t_7]$	p_9
$\sigma_2 = t_1t_2t_4t_5t_6t_4t_5t_6t_4t_3t_7$	$p_8 + p_9$	IC	$p_5 + p_8 + p_9$	NO	p_7	NO	p_9
$\sigma_3 = t_1t_2t_4t_3$	p_7	NO	$p_5 + p_7$		p_7	NO	$p_5 + p_7$

$$1 - halt(a_2, q) = 0 \Rightarrow \beta_1(a_2, \sigma_1) = 0$$
$$|q \ominus q(end(a_2))| = 0 \Rightarrow \beta_2(a_2, \sigma_1) = 0$$
$$n - r = 0 \Rightarrow \beta_3(a_2, \sigma_1) = 0$$

Trace σ_2 leads to the state $q = p_8 + p_9$ producing an improper completion and $badness(a_2, h, \sigma_1) = 0.11$:

$$halt(a_2, q) = 0 \Rightarrow \beta_1(a_2, \sigma_2) = 0$$
$$\beta_2(a_2, \sigma_2) = 1 \cdot \frac{1}{9} \cdot 1 = 0.11$$
$$n - r = 0 \Rightarrow \beta_3(a_2, \sigma_2) = 0$$

Trace σ_3 leads to the state $q = p_7$ producing a no option to complete and $badness(a_2, h, \sigma_2) = 0.33$:

$$\beta_1(a_2, \sigma_3) = 1 \cdot \frac{1}{3} \cdot 1 = 0.33$$
$$1 - halt(a_2, q) = 0 \Rightarrow \beta_2(a_2, \sigma_3) = 0$$
$$n - r = 0 \Rightarrow \beta_3(a_2, \sigma_3) = 0$$

a_3

Trace σ_1 leads to state $q = p_5 + p_9$ producing an improper completion, and $badness(a_3, h, \sigma_1) = 0.11$:

$$1 - halt(a_3, q) = 0 \Rightarrow \beta_1(a_3, \sigma_1) = 0$$
$$\beta_2(a_3, \sigma_1) = 1 \cdot \frac{1}{9} \cdot 1 = 0.11$$
$$n - r = 0 \Rightarrow \beta_3(a_3, \sigma_1) = 0$$

Trace σ_2 leads to the state $q = p_5 + p_8 + p_9$ producing an improper completion and $badness(a_3, h, \sigma_2) = 0.22$:

$$1 - halt(a_3, q) = 0 \Rightarrow \beta_1(a_3, \sigma_2) = 0$$
$$\beta_2(a_3, \sigma_1) = 1 \cdot \frac{2}{9} \cdot 1 = 0.22$$
$$n - r = 0 \Rightarrow \beta_3(a_3, \sigma_2) = 0$$

Trace σ_3 leads to an intermediate state $q = p_5 + p_7$ and $badness(a_3, h, \sigma_3) = 0$:

$$halt(a_3, q) = 0 \Rightarrow \beta_1(a_3, \sigma_3) = 0$$
$$terminal(a_3, q) = 0 \Rightarrow \beta_2(a_3, \sigma_3) = 0$$
$$n - r = 0 \Rightarrow \beta_3(a_3, \sigma_3) = 0$$

a_4 Trace σ_1 leads to state $q = p_7$ which quasi-enables its last transition t_7 producing a quasi-enabled transition error, it is also a terminal state which produces also a no option to complete error. The overall badness is $badness(a_4, h, \sigma_1) = 0.51$:

$$\beta_1(a_4, \sigma_1) = \frac{6}{7} \cdot \frac{1}{2} \cdot 0.43$$
$$halt(a_4, q) = 0 \Rightarrow \beta_2(a_4, \sigma_1) = 0$$
$$\beta_3(a_4, \sigma_1) = \frac{1}{6} \cdot \frac{1}{2} \cdot 1 = 0.08$$

Similarly to the previous trace, trace σ_2 leads to the state $q = p_7$ producing a quasi-enabled transition error and a no option to complete error. The overall badness is $badness(a_4, h, \sigma_2) = 0.37$:

$$\beta_1(a_4, \sigma_2) = \frac{6}{7} \cdot \frac{1}{3} \cdot 1 = 0.29$$
$$halt(a_4, q) = 0 \Rightarrow \beta_2(a_4, \sigma_2) = 0$$
$$\beta_3(a_4, \sigma_2) = \frac{1}{6} \cdot \frac{1}{2} \cdot 1 = 0.08$$

Trace σ_3 leads also to the state p_7 but in this case only a no option to complete error is generated, since all transitions are performed. The badness is $badness(a_3, h, \sigma_3) = 0.25$:

$$\beta_1(a_4, \sigma_3) = 1 \cdot \frac{1}{4} \cdot 1 = 0.25$$
$$halt(a_4, q) = 0 \Rightarrow \beta_2(a_4, \sigma_3) = 0$$
$$n - r = 0 \Rightarrow \beta_3(a_4, \sigma_3) = 0$$

a_5 For computing the badness of a_5 we consider the possibility of performing an additional silent transition during each trace simulation. Trace σ_1 and σ_2 are correct traces, they both lead to state $q = p_9$ and $badness(\sigma_1) = 0$:

$$1 - halt(a_5, q) = 0 \Rightarrow \beta_1(a_5, \sigma_1) = 0$$
$$|q \ominus q(end(a_5))| = 0 \Rightarrow \beta_2(a_5, \sigma_1) = 0$$
$$n - r = 0 \Rightarrow \beta_3(a_5, \sigma_1) = 0$$

Trace σ_3 leads to an intermediate state $q = p_5 + p_7$ and $badness(a_5, h, \sigma_3) = 0$:

$$halt(a_5, q) = 0 \Rightarrow \beta_1(a_5, \sigma_3) = 0$$
$$terminal(a_5, q) = 0 \Rightarrow \beta_2(a_5, \sigma_3) = 0$$
$$n - r = 0 \Rightarrow \beta_3(a_5, \sigma_3) = 0$$

□

♢

4.7.4 Simulated Annealing

Given an unsound workflow net $a \in \mathcal{W}$, the main goal of the PNSA technique is to produce a good set of solutions $S = \{s_i\}_{i=1}^{k} \subseteq \mathcal{W}$ such that each model $s_i \in S$ is similar to a but contains fewer or no errors. S can be considered good if its elements are correct with high confidence and they are not redundant, namely no one element can be considered better than any other one. This latter concept is captured by the notion of dominance and Pareto-set. Fixed the model under analysis $a \in \mathcal{W}$ and the trace simulation $h \in \mathcal{H}$, the following objective functions $f_i : \mathcal{W} \times \wp(\mathcal{T}_R) \to \mathbb{R}$ can be defined: $f_1(x, R) = \mathit{stdist}(a, x)$, $f_2(x, R) = \mathit{bhdist}(a, x, h, R)$ and $f_3(x, R) = \mathit{badness}(x, h, R)$. The objective functions can be grouped into a unique function $\bar{f} : \mathcal{W} \times \wp(\mathcal{T}_R) \to \mathbb{R}^3$ such that $\forall x \in \mathcal{W}$ identifies the triple $(\mathit{stdist}(z, x), \mathit{bhdist}(z, x, h, R), \mathit{badness}(x, h, R))$.

Definition 4.33 (Dominance). A solution $x \in \mathcal{W}$ dominates a different solution $y \in \mathcal{W}$ if it is better in at least one objective and equivalent in the remaining ones. The relation can be formally stated as

$$\mathcal{D}(R) = \{(x, y) \mid \exists x, y \in \mathcal{W} \,.\, \forall i \in [1, 3] \,. \\ f_i(x, R) \leq f_i(y, R) \wedge \exists j \in [1, 3] \,.\, f_j(x, R) \neq f_j(y, R)\}$$

A pair $(x, y) \in \mathcal{D}(R)$ is denoted as $x \prec_R y$ or $x \prec y$ when R is clear from the context, similarly a pair $(x, y) \notin \mathcal{D}(R)$ is denoted by $x \not\prec_R y$ or $x \not\prec y$.

Definition 4.34 (Pareto-set). Two not comparable solutions $x, y \in \mathcal{W}$ such as $x \not\prec y$ and $y \not\prec x$ are said to be mutually non-dominating. A Pareto-set is a set of mutually non-dominating solutions, formally $\mathcal{P} = \{x_i\}_{i=1}^{n}$ is a *Pareto-set* with respect to R if and only if $\forall i, j \in [1, n] \,.\, x_i \not\prec_R x_j$.

Definition 4.35 (Pareto-front). The Pareto-front $\mathcal{F}(R) \subseteq \mathbb{R}^3$ is the set of all points in the objective space that correspond to Pareto-optimal solutions. The Pareto-front with respect to R is defined as $\mathcal{F}(R) = \{\bar{f}(x, R) \mid \exists x \in \mathcal{W} \,.\, \forall y \in \mathcal{W} \,.\, y \not\prec_R x\}$.

In MOSA applications the objective functions are composed in some way to obtain a unique fitness function to be minimized. In dominance-based MOSA the fitness is defined in terms of the gap between the objective values of a solution and a set of optimal values of the Pareto-front. Since the true Pareto-front $\mathcal{F}$ is generally unknown, the fitness function is based on a finite approximation of it $F \subseteq \mathbb{R}^3$, called estimated Pareto-front.

Definition 4.36 (Fitness measure). Let $F \subseteq \mathbb{R}^3$ a finite estimation of the Pareto-front $\mathcal{F}$, $x \in \mathcal{W}$ a solution and $R \subseteq \mathcal{T}_R(a)$ a set of traces of a. The fitness function $\mathit{fit} : \mathbb{R}^3 \times \mathcal{W} \times \mathcal{T}_R \to \mathbb{R}$ is defined as the number of points in F that are better than $\bar{f}(x, R)$. Formally

$$\forall F \subseteq \mathbb{R}^3 \,.\, \forall x \in \mathcal{W} \,.\, \forall R \in \mathcal{T}_R \,.\, \mathit{fit}(F, a, R) = |\{v \in F \mid v < \bar{f}(x, R)\}|$$

The estimated Pareto-front F can be initialized in different ways and updated during the computation for increasing its quality. In this application F is a function of the Pareto-set $\mathcal{P}$ under construction: it is initially empty and incrementally populated with new values as long as new solutions are added to $\mathcal{P}$. More specifically, when a new solution is added to $\mathcal{P}$ its corresponding point in the objective space is added to F together with some other artificial points in its neighborhood.

In the annealing procedure, a new solution $y \in \mathcal{W}$ is accepted in place of the current one $x \in \mathcal{W}$ on the basis of the fitness difference $\Delta_{fit}(y, x, R)$ defined as

$$\Delta_{fit}(y, x, R) = \frac{1 + fit(\tilde{F}, y, R) - fit(\tilde{F}, x, R)}{1 + |F|} \qquad \tilde{F} = F \cup \left\{\bar{f}(x, R), \bar{f}(y, R)\right\}$$

More specifically, a new solution $y \in \mathcal{W}$ is accepted in place of the current one $x \in \mathcal{W}$ with a probability equal to $min\,\{1, exp\,(-\Delta_{fit}(y, x, R)/T(i))\}$, where $T(i)$ is a monotonically decreasing function indicating the temperature for the iteration i of the annealing procedure. This ensures that if the new candidate solutions y is better than the current one x (it moves towards the estimated Pareto-front), it is always accepted in place of x; otherwise, it is still accepted with a probability that decreases with the temperature. The temperature allows one to compensate the concept of confidence: at the beginning the confidence of a solution is relatively small, hence in this phase a solution can be discarded in place of a new one which is not strictly better, because the set of traces chosen so far can be not particularly significative; conversely, when the confidence about a solution increases it is less likely discarded in place of a worsen one, because the temperature is decreased as well. Let us notice that if the candidate solution y has the same fitness of the current solution x, we maintain the current solution since this has also been tested against some other trace sets. This is captured by the notion of *confidence* of a solution, which indicates how many annealing iterations a solution has survived through. The more iterations a solution survives through, the more the confidence increases that this is a good solution.

Example 4.37. Let us assume an estimated Pareto-front of $\{p_1 = (0, 0, 0.41), p_2 = (5, 0.52, 0.43), p_3 = (9, 0, 0), p_4 = (11, 0, 0.71)\}$ for our working example in Fig. 4.11. The structural similarity, the behavioural similarity and the badness of nets a_3, a_4, and a_5 have been computed in Ex. 4.25, Ex. 4.30, and Ex. 4.32, respectively. The corresponding points in the objective space are $o_3 = (1, 0, 0.33)$, $o_4 = (4, 0.34, 1.13)$, and $o_5 = (5, 0, 0)$. Therefore, net a_3 dominates points p_2 and p_4, while net a_4 dominates only point p_2, and net a_5 dominates points p_2, p_3, and p_4. It results that a_5 has higher probability of being accepted in place of the other two. □

The annealing procedure is reported in Lis. 4.4. It requires in input (1) the original model $a \in \mathcal{W}$, (2) the Pareto-set $\mathcal{P}$ of the solutions found so far, (3) a parameter $c \in \mathbb{N}$ representing the desired confidence of the result, (4) the maximum number of iterations $m \in \mathbb{N}$, (5) the number of sample traces considered at each iteration $k \in \mathbb{N}$, (7) a monotonically decreasing function $T : \mathbb{N} \to \mathbb{R}$ representing a cooling schedule such that $T(i)$ is the temperature used at the iteration $i \in [1, m]$. The algorithm produces in output a list of at most n solutions of decreasing confidence where the first element has confidence at least c when the algorithm converges before reaching the maximum number of iterations m.

The main steps of the algorithm can be summarized as follows:

1-5 Initialization of the main variables used in the algorithm. The estimated Pareto-front F is initialized by the procedure ESTIMATE-PARETO-FRONT by using the given Pareto-set $\mathcal{P}$. As mentioned above, in order to increase the size of F, some points in the neighborhood of the computed ones are artificially added. The solution list L is originally empty, its elements are always ordered by their decreasing confidence and when the list becomes full, elements with lower confidence are discarded. Net x is initial current solution obtained by combining one or more elements chosen from $\mathcal{P} \cup \{a\}$, while c_x is its confidence. Finally i is the number of iterations of the main cycle performed so far.

6-22 The main loop of the procedure. The loop terminates when the maximum number of iterations m has been reached or the confidence of the current solution is greater than the desired one c.

7 Generation of a perturbation $y \in \mathcal{W}$ of x. The perturbation is obtained by a small sequence of edit operations that are chosen at random or selected exploiting the solutions stored in L.

8 Procedure SELECT-TRACES chooses at random or with a better strategy a finite representative set of k traces $R \subseteq \mathcal{T}_R(a)$, considering also the set of errors $errors(a)$ of the original model. For instance if $errors(a)$ is small, ensure $errors(a) \subseteq R$.

9-12 Computation of the objective functions of x and y with respect to R and of the sets of errors and fixes found in the solutions under analysis for later use. A solution x with new errors $errors(x) \cap errors(a) \neq \varnothing$ is not discarded at priori because it may contain a partial fix to the target error set $errors(a)$.

13-15 Computation of the difference $\Delta_{fit}(y, x, R)$ between the fitness of y and x using the available estimated Pareto-front F, and of the acceptance probability. The perturbed solution y is accepted in place of the current one x with a probability less than $min\{1, exp(-\Delta_{fit}(y, x, R)/T(i))\}$, where $T(i)$ is the temperature for the current iteration i.

16-21 Choice between the two solutions. If the computed value p is less than a random value between 0 and 1, then the perturbed solution y is accepted in place of the current solution x. If y is accepted, x is stored in L on the basis of its confidence c_x and y becomes the current solution. Otherwise, y is discarded and the confidence about x is increased.

23-24 At the end the current solution x is stored in L and list L is returned.

The PNSA technique consists of several runs of the annealing procedure in order to incrementally construct a Pareto-set formed by optimal non-dominated solutions with high confidence. At each run of the annealing procedure, the first element of L is added to the current Pareto-set, depending on the resulting confidence and the presence of new errors, which is used by the next run. The procedure can be stopped as soon as the Pareto-set satisfies the end user. The details of the overall algorithm are reported in Lis. 4.5.

1-2 Initialization of the main variables used in the algorithm. The Pareto-set of solutions S under construction is initialized as an empty set, while the

Listing 4.4 The annealing procedure

input: The original model $a \in \mathcal{W}$ expressed in WFNs.
input: The Pareto-set $\mathcal{P}$ of solutions found so far.
input: The desired confidence $c \in \mathbb{N}$ of the result.
input: The maximum number of iterations $m \in \mathbb{N}$.
input: The number of sample traces $k \in \mathbb{N}$ to consider at each iteration.
input: A monotonic decreasing function $T : \mathbb{N} \to \mathbb{R}$ representing the cooling schedule, such that $\forall i \in N \,.\, T(i)$ is the temperature at iteration i.
output: A list L of at most n solutions with decreasing confidence.

$L \leftarrow$ ANNEALING$(a, \mathcal{P}, c, m, k, T)$

```
 1  F ← ESTIMATE-PARETO-FRONT(P)
 2  L ← ∅
 3  x ← GENERATE-INITIAL-SOLUTION(P ∪ a)
 4  c_x = 0
 5  i = 0
 6  while c_x < c ∧ i ≤ m do
 7      y ← PERTURBATE(x)
 8      R ← SELECT-TRACES(T_R(a), errors(a), k)
 9      (o_x, e_x) ← COMPUTE-OBJECTIVE-FUNCTION(x)
10      errors(x) ← errors(x) ∪ e_x
11      (o_y, e_y) ← COMPUTE-OBJECTIVE-FUNCTION(y)
12      errors(y) ← errors(y) ∪ e_y
13      Δ_fit ← COMPUTE-DELTA-FIT(o_y, o_x, F)
14      p ← min{1, exp(−Δ_fit / T(i))}
15      r ← GENERATE-RANDOM(0, 1)
16      if p > r then
17          L ← STORE(L, (x, errors(x)), c_x)
18          x ← y
19      else
20          c_x ← c_x ∪ |R|
21      end if
22  end while
23  L ← STORE(L, (x, errors), c_x)
24  return L
```

cooling schedule T is initialized considering a maximum and a minimum temperature and the number of step performed by each annealing run.

3-6 The main loop of the algorithm. Until the desired number of solutions n is not generated, a new run of the annealing procedure is performed and the head of the returned list is added to the current Pareto-set of solutions.

Listing 4.5 The PNSA procedure

input: The original model $a \in \mathcal{W}$ expressed in WFNs.
input: The desired confidence $c \in \mathbb{N}$ of the result.
input: The maximum number of iterations $m \in \mathbb{N}$ for each annealing.
input: The number of sample traces $k \in \mathbb{N}$ to consider at each annealing iteration.
input: The maximum size of the solution list $n \in \mathbb{N}$.
input: The initial temperature $t_i \in \mathbb{R}$.
input: The freezing temperature $t_f \in \mathbb{R}$.
output: A set of n solutions $S \in \mathcal{W}$.

$S \leftarrow \textsc{Pnsa}(a, c, m, k, n, t_i, t_f)$
1 $S \leftarrow \varnothing$
2 $T \leftarrow \textsc{Generate-Cooling-Schedule}(t_i, t_f, k)$
3 **while** $|S| < n$ **do**
4 $\quad L \leftarrow \textsc{Annealing}(a, S, c, m, k, T)$
5 $\quad S \leftarrow S \cup \textsc{Head}(L)$
6 **end while**
7 **return** S

4.8 Experimental Results

We implemented the PNSA algorithm in a prototype Java tool. This tool imports an unsound workflow net in LoLA format [93], and executes on it the soundness check in Lis. 4.2. The result of the soundness check and the user parameters trigger the PNSA algorithm. At each iteration i of the annealing procedure, the tool performs at most k prioritized random walks on the state-space to extract the sample traces of set R_i. More precisely, for each trace, the state-space is traversed backwards starting from a final state and by prioritizing those transitions that have been visited the least, until the initial state is reached. Correct traces are extracted starting from q_e; erroneous traces are extracted starting from a final state that led to an error (available from the soundness check). The output of the tool is a set of solutions in LoLA format, which is limited by the maximum response time or by the maximum number of solutions set by the user.

We used the tool to fix a sample of 152 unsound nets drawn from the BIT process library [94]. This library contains 1,386 BPMN models in five collections (A, B1, B2, B3, C), out of which 744 are unsound. We converted these 744 models into workflow nets and filtered out those models that did not result in valid workflow nets (e.g. those models that had multiple output places). In particular, we only kept models with up to two output places, and when we found two output places we merged them in a single output place. Since this operation may introduce a lack of synchronization, we discarded models with more than two output places to minimize the impact of such artificial errors. As a result, we obtained 152 models, none of them from collection C. Tab. 4.1 reports some information about the obtained input models, such as their average and maximum dimension in terms of node numbers and the average and maximum found errors.

Collection	No. models	Avg/Max nodes	Avg/Max errors
BIT A	48	46.54 / 129	2.21 / 5
BIT B1	22	29.55 / 87	2.55 / 6
BIT B2	35	22.86 / 117	2.54 / 9
BIT B3	47	20.38 / 73	2.77 / 9

Table 4.1. Experimental results.

We set the maximum number of final solutions to 6, the desired confidence to 100, the maximum number of iterations for each annealing run to 1,000, the initial temperature to 144, the maximum size of each set of sample traces to 50, and all the parameters for behavioral distance and badness to 1. After the experiment each solution was checked for soundness. The tests were conducted on a PC with a 3GHz Intel Dual-Core x64, 3GB memory, running Microsoft Windows 7 and JVM v1.5. Each test was run 10 times by using the same random seed (to obtain deterministic results) and the execution times were averaged. The results are reported in Tab. 4.2.

Collection	Avg/Max nodes	Avg/Max errors	Avg Error reduction	Sound models	Avg/Max str. distance [cost]	Avg/Max time[s]
BIT A	45.85 / 129	0.74 / 20	73.6%	85.7%	4.25 / 39	16.52 / 171.43
BIT B1	27.18 / 87	1.07 / 29	75.6%	82.9%	5.02 / 91	8.75 / 100.45
BIT B2	21.79 / 118	0.69 / 9	84.9%	87.3%	3.47 / 47	12.08 / 217.91
BIT B3	21.42 / 118	1.26 / 24	72.4%	79.6%	4.62 / 61	10.85 / 156.93

Table 4.2. Experimental results.

The error-reduction rate of a solution is the difference between the number of errors in the initial model and the number of errors in the solution, over the number of errors in the initial model. Tab. 4.2 shows the average error-reduction rate for all solutions of each collection, which leads to an average error-reduction of 76.6% over all four collections. This indicates that the algorithm is able to fix the majority of errors in the input models. Moreover, the structural distance of the solutions is very low (4.34 on average using a cost of 1 for each edit operation). Thus, the solutions are very similar to the original model. These solutions are obtained with an average response time of 11s. Despite the large response time in some outlier cases (218s), most solutions are behaviorally better than their input models (83.9% are sound) and at least one sound solution was found for each input model. Very few cases are worse than the input model (e.g. 29 errors instead of 6 errors in the input model). This is due to the fact that we did not fine-tune the annealing parameters based on the characteristics of each input model (e.g. if the number of annealing iterations is too low w.r.t. the number of errors in the input model, a solution could contain more errors than the input model).

4.9 Summary and Concluding Remarks

This chapter deals with the verification and the automatic correction of process models. For this purpose, it initially introduces the WFNs language, a Petri nets variant tailored for modeling and analyzing processes in the context of workflow management systems. On such language the notion of soundness has been defined which tries to capture the models that will provide a good run-time behaviour. The original notion of soundness proposed in literature is formally stated and some problems about it are exposed. This results in the definition of a revised soundness formulation whose properties are orthogonal. On the basis of such notion, a refined soundness check procedure is illustrated which verifies the presence of the desired properties on a given model. The proposed procedure is slightly different from the traditional one, because its primary aim is to capture a set of additional information about the found errors, in order to exploit them during a subsequent repair phase. The automated model repair problem is then introduced using the WFNs as the reference formal language for encoding the models to fix.

Finally, a novel optimization technique, called PNSA, is proposed which given a model containing some errors and the information collected during the verification phase, tries to find a set of solutions which are structurally and semantically similar to the original model but with fewer errors. A set of experimental results are also reported which illustrates the application of the proposed techniques to a set of real process models encoded as workflow nets.

This chapter focuses on the current modeling practice characterized by the use of unstructured PMLs supporting the free-design paradigm. Conversely, the following chapter proposes an alternative modeling approach based on the adoption of a structured PML able to impose some constraints during design and to produce models with a block-structured control-flow. This approach allows one to exclude the presence of certain errors by construction, that are otherwise both difficult to spot and fix without rethinking the whole model.

5

Towards Structured Process Modeling Languages

PMLs are usually classified as *unstructured* due to their graph-oriented syntax and token-based semantics inspired by Petri nets. They allow a free composition of constructs without worrying much about the type or position of the connected elements. Using an unstructured PML, a more or less structured model can be built: a sub-graph of a model is considered *structured* when its constructs are properly nested and correctly matched to produce a block with one entry and one exit point, as discussed for instance in [17, 95]. Research concerning the quality of process models confirms that structured forms should be preferred to unstructured ones, because they improve model comprehensibility and reduce the probability to accidentally introduce errors inside models [96, 97]. These effects should not surprise: firstly, the presence of a structured form is an easy-to-verify syntactical property that can guarantee the presence of desirable semantic properties which can be otherwise hard to prove; secondly, structured forms enhance modularity: a key property in any complex design activity performed by humans.

PMLs can also be classified by considering the set of provided constructs into two main categories: data-flow oriented (DFO) and control-flow oriented (CFO). A *DFO language* describes a process by explicitly representing data dependencies among its constituent components. The declared data relations can be used for transparently exploiting parallelism without requiring explicit synchronization. On the contrary, a *CFO language* focuses on the execution order of components, leaving data dependencies almost implicit. Even if the DFO paradigm provides higher modularity, the CFO one has been historically adopted for designing PMLs because the explicit representation of all data dependencies among components and the massive parallelism when abused can become a drawback in process design. Nevertheless, CFO languages are apparently simpler than DFO ones, at least because data aspects can be ignored at early design stages. This over simplification is paid at later design stages and during implementation, when data-flow relations shall be finally integrated in order to obtain an executable system. Integrating DFO abstractions in CFO languages is therefore unavoidable and at the same time challenging due to the commonly adopted CFO constructs.

The aim of this chapter is to promote the development of a new generation of PMLs that adopt a structured design approach enhanced with DFO abstractions. For this purpose, the chapter starts by justifying the adoption of a structured

control-flow by exposing the problems that a lack of structure may determine, then it introduces a novel PML called NestFlow, which provides block-structured control-flow constructs combined with asynchronous message passing abstractions.

The process models treated in this chapter are expressed using different PMLs widely discussed in the previous chapters, such as BPMN, YAWL, and WFNs. The priority is given to BPMN because it seems to have the more intuitive notation, while YAWL and WFNs are used when a more formal semantics is required. Anyway, except for some processes including message exchange, the discussed BPMN models can be easily mapped to YAWL or even to WFNs, since no advanced constructs are used.

The execution of a process instance is represented as a sequence of steps denoted using the following notation: each step is represented by the symbol $\rightarrow$, while the overall state of threads is denoted using construct and component identifiers. In particular, a component will compare at least two times in a sequence: when the it starts and when it terminates. If a component A reads a variable x in its scope when starts, then it is annotated as $x|A$. Similarly, if A writes a variable y when terminates, then it is denoted as $A|y$. In case multiple variables are read or written at the same time, they are enclosed in curly braces, e.g. $\{x, y\}|A$ means A starts by reading x and y. Multiple threads of control can be created and destroyed during a run: in such case concurrent executions are denoted within parenthesis and separated by commas. For instance, the sequence $A \rightarrow A|x \rightarrow (x|B, C) \rightarrow (B, C|y) \rightarrow B|z$ stands for: the component A starts its execution, then it ends by writing the variable x, subsequently B starts in parallel with C by reading the value of x, then C terminates by writing the variable y, while B is still running, and finally B concludes by writing the variable z. The step symbol $\rightarrow$ can also be annotated with a condition and a possible result of its evaluation for stating that its execution is determined by a system choice.

The notion of soundness and its related errors have been widely discussed in the previous chapters: here the concepts of no option to complete, improper completion, and dead transitions are informally applied also to models that are not workflow nets, but are expressed for instance in BPMN or YAWL. This is reasonable, because the stated properties can be proven by mapping the models to WFNs abstracting away undesired details about data.

The remainder of this chapter is organized as follows: Sec. 5.1 summarizes some research efforts which support the adoption of a structured paradigm in process design. Sec. 5.2 discusses the most severe errors that can be introduced in a process model when an unstructured approach is adopted. Sec. 5.3 exposes the weakness of some arguments against structured modeling approaches that have the negative effect to prevent any further investigation in this direction. Sec. 5.4 discusses the key rule of structure in process modeling, while Sec. 5.5 highlights some modularity issues of unstructured PMLs. Sec. 5.6 introduces NestFlow by discussing its main features. The NestFlow language constructs are deeply analyzed in Sec. 5.7 and Sec. 5.8. Finally the formal semantics of NestFlow is presented in Sec. 5.9.

5.1 Related Work

In [17] Kiepuszewski et al. investigate the expressiveness of unstructured workflow languages with some syntactical restrictions. They show that some well-behaved unstructured models cannot be transformed into structured ones. Based on this study, Liu and Kumar in [95] analyze unstructured workflows introducing a taxonomy of unstructured forms and determining which ones have an equivalent structured form. In both contributions the authors consider structured languages less expressive than unstructured ones; however, their studies concern only control-flow forms abstracted from other language aspects; hence, they do not exclude the existence of fully-fledged structured PMLs. In [96, 98] Mendling and Laue investigate the importance of structuredness for obtaining correct models: they introduce two different metrics that capture the degree of unstructuredness and relate these metrics to the probability of finding an error. They conclude that structuredness is an important property for increasing the quality of models. Moreover, Reijers and Mendling in [97] analyze modularity and conclude that such property has positive effects on the comprehensibility of large-scale business process models.

These effects are also confirmed by Vanhatalo et al. in [99, 100] where they present a parsing technique, called Refined Process Structure Tree (RPST), useful for detecting structured subgraphs inside generic models. RPST has several applications, e.g. it can be used as a first step for translating a graphical process model into a low-level executable specification, which can be directly interpreted by a PAIS engine. A first analysis about modularity and concurrency of PMLs in the context of business process modeling can be found in [19, 20], where Combi et al. investigate some critical design problems that emerge when a graphical modeling language is obtained by coupling unstructured routing constructs with shared variables and message passing primitives in order to provide all the abstractions needed to produce an executable process model.

NestFlow embodies many principles of the actor model [101]: the behaviour of a component is history sensitive, a component is made of other component instances statically declared or created at run-time, and components interact through buffered asynchronous communications. A component encapsulate both data and logic providing a mechanism to protect the implementation against improper operations. Therefore, information is local to each component and it has to be explicitly transferred in order to be known by any other agent. Even if some other similarities can be found, NestFlow is not a strict implementation of the actor model. Indeed, while in the latter there is no assumption about the concurrent execution of agents, that are intended free to run in parallel, NestFlow offers specific control-flow constructs for limiting parallel executions whenever necessary.

An interesting general-purpose programming language for structured parallel programming is proposed by Bläser in [102]. Such language also offers a minimal graphical notation for components. The aim of the proposed component language is to support hierarchical decomposition of components without explicit pointers, unrestricted symmetric polymorphism, and communication-based interactions. A component can expose multiple independent interfaces that declare what connections are offered and required. A component can contain any statically or dynamically created component instances. The component logic and the connections

among components are fully encapsulated and exclusively managed by the surrounding component. Component can run in parallel respecting the communication protocols specified by the interfaces.

In [103] the authors propose the use of DFO languages as coordination languages [9], in which fine-grained operations on data are implemented with an underlying general-purpose programming language. At the same time they emphasize the need for CFO constructs during design, in order to prune away fine-grained parallelism, toward a more coarse-grained execution. In [104] the authors address the problem of modeling typical control-flow constructions in scientific workflows, such as loop and switch-case statements, using only data-flow relations. This operation can quickly lead to complex workflows, hence they propose a solution based on the encapsulation of these generic constructions in workflow templates. In [105] the authors put DFO languages one step forward introducing the notion of *data-flow process networks*, an evolution of the *Kahn process networks* [106] in which components are not single instructions or simple functions, but more or less sophisticated processes that communicate through message passing. This model represents the theoretical foundation of some scientific workflow systems, like Kepler [37]. Conversely, the introduction of DFO abstractions in CFO languages has received less attention in literature.

◊

5.2 Common Pitfalls in Unstructured PMLs

Many kind of errors may be accidentally introduced in a process model during the design activity, but the most subtle ones concern unwanted interaction patterns that may occur among two or more concurrent entities. These errors are considered subtle because they are difficult to spot, and usually they cannot be corrected in any obvious way. This section aims to expose the root causes that lead to this kind of errors and explain how structured compositions help in avoiding them from the beginning. Unfortunately, unstructured forms cannot be entirely avoided due to a lack of expressiveness of the available control-flow oriented PMLs and they are commonly accepted as a necessary evil.

The BPMN process model in Fig. 5.1 will be used through the entire section to ease the discussion. It does not include real tasks or data, because the focus is on the expressed logic; nevertheless, it contains sufficient information to be executed on a real system or translated into a different language, like YAWL. Task A updates the variable x that is read by B and F; task F also takes the value produced by E that is temporarily stored in y. For simplicity, it is supposed that F uses x and y to compute the expression $z \leftarrow y/x$. B acts as a monitoring activity that, given the value in x, decides when to leave the loop $\{u_1, A, s, B, v_1\}$ by placing a truth value in b. Tasks D or E followed by F are performed in parallel with B at least once before the process can complete. When B decides to exit from the loop, it is assumed that C loads the current value in z to permanently store it into a database. Such value can be produced by D or F depending on the choice v_2. The described unstructured model cannot be considered well-behaved, because it contains several issues. Each of them is used to introduce the general problems that arise with an unstructured control-flow design.

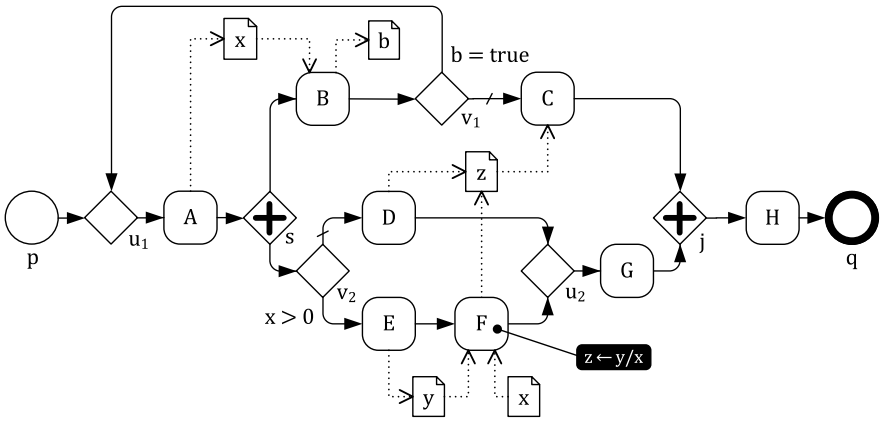


Fig. 5.1. An unstructured process model containing some subtle errors. The model is expressed in BPMN enhanced with variable names x, y, z, b, construct identifiers p, q, u_1, v_i, v_2, s, j, and explicit conditions that usually are not part of the graphical notation.

(P1) Control-Flow Entanglement

In many unstructured PMLs implicit fine-grained data-flow dependencies require explicit fine-grained synchronization points; these points shall be connected together with control-flow arcs that often break existing structured forms. For example, in the model of Fig. 5.1 there is no guarantee that C reads exactly the last value produced by D or F. Eventually, the following events occur:

$$\begin{aligned} p \to u_1 \to A \to A|x \to s \to (x|B, v_2) &\xrightarrow{x \leq 0} (B, D) \to (B|b, D) \to \\ &\to (v_1, D) \xrightarrow{b = false} (z|C, D) \to (C, D) \to (j, D) \to (j, D|z) \to \\ &\to (j, u_2) \to (j, G) \to (j, j) \to H \to q \end{aligned}$$

Moving C after u_2 or j is not a viable alternative, hence certain control-flow connections have to be added for synchronizing D and F with C. Such connections inevitably break the structured block $\{v_2, D, E, F, u_2\}$.

(P2) Undesired Tokens

With an unstructured control-flow design, it can be difficult to track and deal with tokens left in different places of a model during its execution. For example, the model in Fig. 5.1 suffers of an improper completion because the loop $\{u_1, A, s, B, v_1\}$ potentially produces more than one token that flows through the bottom branch of the And-Split s, finishing their run in j. Nevertheless, only one token in the subgraph $\{v_2, D, E, F, u_2\}$ is correctly synchronized in j before exiting. In YAWL the remaining tokens can be withdrawn by surrounding the subgraph $\{v_2, D, E, F, u_2, j\}$ with a cancellation region enabled by H. BPMN can simulate a cancellation region by enclosing the subgraph into a sub-process with a boundary exception. However, these solutions are far from being easy to apply, especially if the subgraph has many entry and exit points due to the already mentioned synchronization points.

(P3) Unreliable Invariants

With an unstructured design approach one cannot exclude that multiple tokens flow in the same sequential branch making invariants hard to state and preserve. As a result, the value of a variable cannot be assumed to remain the same between the execution of two sequential steps. Reasoning about the correctness of an unstructured model with free to flow tokens becomes soon a non-trivial activity. For example, F in Fig. 5.1 belongs to a branch guarded by $x > 0$, but one cannot exclude that F computes $z \leftarrow y/x$ with $x = 0$. Indeed, the following trace can occur:

$$\begin{aligned} p \to u_1 \to A \to A|x \to s \to (x|B, v_2) &\xrightarrow{x > 0} (B, E) \to (B|b, E) \to \\ &\to (v_1, E) \xrightarrow{b = true} (u_1, E) \to (A, E) \to (A|x, E) \xrightarrow{x = 0} (s, E) \to \\ &\to (s, E|y) \to (s, \{x, y\}|F) \end{aligned}$$

In such case F is executed with an erroneous input $x = 0$, despite it is placed on a branch guarded by the condition $x > 0$.

5.3 Myths Surrounding PMLs

This section discusses some common misconceptions about PMLs that seem to be so widespread to prevent any further investigation towards structured modeling approaches. The main concepts of the section are exemplified through the model in Fig. 5.2. This model is extracted and adapted from [17], it is formed by a starting place p, an end place q, two loops $\{u_1, A, B, v_1, s_1, C, j_1\}$ and $\{u_2, H, G, v_2, j_2, F, s_2\}$ that run in parallel, two intermediate tasks D and E and two trailing tasks I and L. The main loops mutually synchronize each other at every iteration by means of the branches $\{s_2, D, j_1\}$ and $\{s_1, E, j_2\}$. The loops are also guarded by the same condition $\varphi(\bar{x})$ stated over a set of variables $\bar{x}$ represented as a tuple. The condition has been left unspecified because it is not relevant for the discussion, it may be for example $x < y$ and in such case $\bar{x} = \langle x, y \rangle$. The model seems better than the previous one in Fig. 5.1: indeed, it can be easily mapped to a workflow net and proved to be sound.

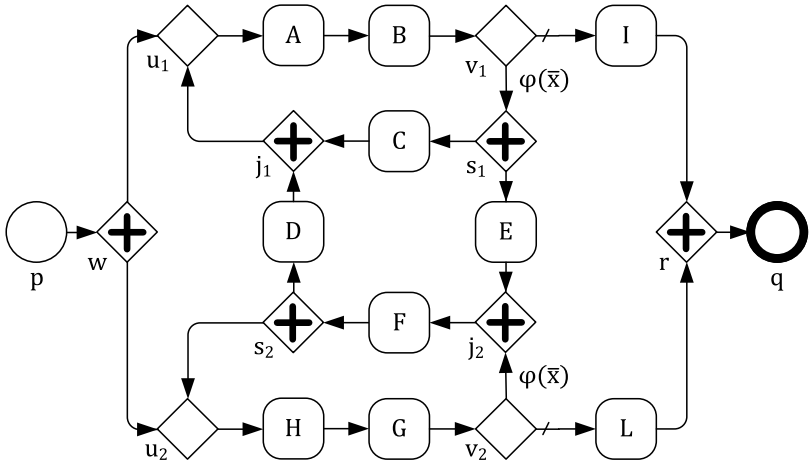


Fig. 5.2. A well-behaved unstructured process model expressed in BPMN.

(M1) The Myth of Expressiveness

Executable unstructured BPMLs are more expressive than structured ones in terms of definable models. This myth is rooted on the elusive notion of structured PML that lacks of an accepted definition or even a reference implementation. This point of view is also supported by mathematical proofs [17]. Actually, these proofs are very weak because they define what is a structured PML starting from an existing unstructured one by adding further syntactical restrictions. Moreover, the considered languages are often so simplified that can be hardly considered executable; they lack of any construct to declare and manipulate data. These proofs usually show that there exist well-behaved unstructured models, as the one in Fig. 5.2, that cannot be expressed with a structured control-flow. The used argument is

fairly trivial since any language with additional constraints is likely to become less expressive, at least if the added constraints are real constrains that affect the language in some way. These kinds of proofs are hard to state on real-world languages, because the structured language at hand may be sufficiently expressive to define an interpreter able to simulate the behavior of the original model step-by-step by encoding unstructured forms as internal data structures.

(M2) The Myth of Soundness

Well-behaved models can be easily distinguished from erroneous ones using the available validation methods. This myth is fostered by the increasing advances in verification methods [88] that can be used to assist the design activity. However, excluding some special cases, validation methods are feasible only if they are applied to an abstraction of the actual executable model, namely an approximation obtained discarding details *considered* irrelevant for the analysis. Whenever the validation method finds an error, it provides a proof of the problem, e.g. a trace that can reproduce the fault in the original model. On the contrary, a successful validation can increase the confidence about model correctness, but it cannot exclude the presence of errors. For example, the model in Fig. 5.2 is sound with respect to the usual notion of soundness adopted for workflow nets [15], but it can easily reach a deadlock condition during its execution: at least one variable $x_i \in \bar{x}$ changes its state, otherwise $\varphi(\bar{x})$ is always false or always true and the loops never execute or never terminate. If the set of variables $\bar{x}$ changes, there exists at least one component t that update it; in particular, a deadlock can be reached for any $t \in \{A, B, C, D, E, G, H\}$. For instance, let $t = A$, the following trace ends with a deadlock:

$$\begin{aligned} p \to w \to (u_1, u_2) \to (A, u_2) \to (A, H) \to (A, G) \to (A, v_2) \xrightarrow{\varphi(\bar{x}) = true} (A, j_2) \to \\ \to (A|x_i, j_2) \to (B, j_2) \to (v_1, j_2) \xrightarrow{\varphi(\bar{x}) = false} (I, j_2) \to (r, j_2) \end{aligned}$$

In this case the passed soundness check may generate false expectations about model correctness.

(M3) The Myth of Refactoring

Any unstructured process model can be refactored in a better one, not necessarily structured. As a consequence no structured PMLs are necessary. This myth is rooted on the questionable assumption that an unstructured PML offers more design freedom because it imposes less construction rules. Actually, unstructured PMLs lack of modularity: hence, certain transformations are far from being easy to perform and sometimes they are even impossible. Let us consider a decomposition similar to the one proposed in [19] for the model in Fig. 5.2: the main process is substituted with a new process P with tasks $\{Q, R, I, L\}$, where the sub-processes $Q = \{u_1, A, B, v_1, s_1, C, j_1, E\}$ and $R = \{u_2, H, G, v_2, j_2, F, s_2, D\}$ encapsulate the two main loops. This is an interesting decomposition, because during design any model is likely to grow in size as more details are added, and the creation of subprocesses, without altering the original behavior of the model, becomes a common

operation. After the Q/R decomposition the two internal loops need to be synchronized in some way for preserving the original semantics. Message passing seems the most suitable abstraction to solve the problem but it is only partially or not supported at all in unstructured PMLs, probably because it cannot be easily integrated with the adopted language constructs. For instance, BPMN explicitly denies the use of message passing inside the same process: a message flow can only be used to connect two separate pools, i.e. for representing interactions among processes that belong to different participants (organizations). Anyway, in [7] the author suggests to not follow the standard during design, and use message passing inside a process whenever needed, even if such practice is not well defined. Another approach to model the communication between R and Q is improperly placing them into two different pools, obtaining the model in Fig. 5.3.

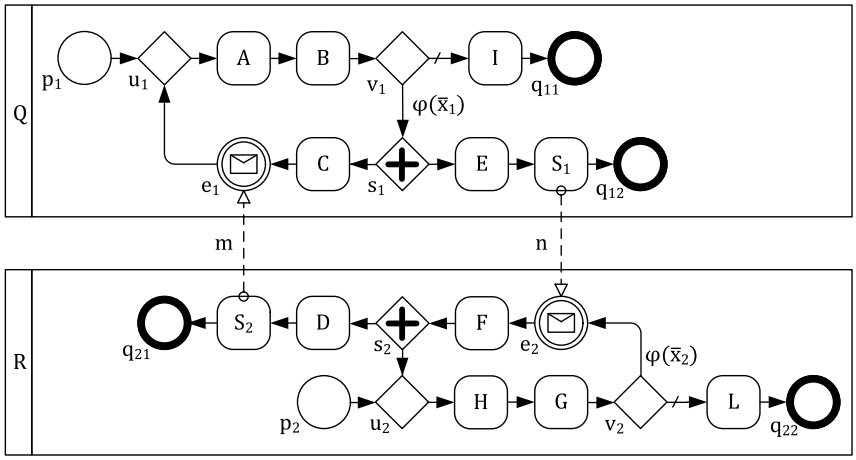


Fig. 5.3. Possible decomposition for the process in Fig. 5.2.

However, such approach does not solve the problem in the general case because the resulting main process depicted in Fig. 5.4 still need message passing, but its components cannot be placed in different pools.

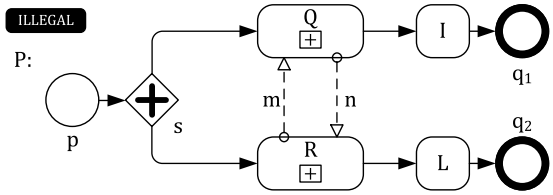


Fig. 5.4. The new main process obtained by combining the two sub-processes Q and R that communicate through message passing.

In other languages like YAWL asynchronous message passing is not supported at all: hence, the decomposition can only be obtained by escaping the language limits, e.g. by writing and reading a database content, making synchronization tricky and verification methods even less effective.

5.4 The Key Rule of Structure

Graphical PMLs can be extremely effective in describing the interaction among concurrent entities that can be naturally expressed in parallel branches. Moreover, they can boost up process logic comprehensibility by exploiting the innate ability of humans in recognizing recurring patterns. Surprisingly, many graphical PMLs do not take into account such aspect sacrificing it in name of free composition. With such paradigm, end-users are free to place language constructs anywhere they want and connect them using arcs, resembling what they can do with a pencil on paper: it seems that the ease of drawing on paper is more important than the ease of reading, despite that models are commonly built using editors which include also verification tools and refactoring operations.

Graphical PMLs that adopt a free-composition approach do not prevent one to make recurring patterns nearly unrecognizable by humans. Let us consider for example the workflow net z_1 in Fig. 5.5a. The net implements a very simple logic: there are three transitions labeled A, B, and C, if A runs first, then B is mandatory; conversely, if B is chosen, then one can choose between A and C. Both nets z_2 and z_3 in Fig. 5.5.b and Fig. 5.5.c, respectively, performs the same logics of z_1. In particular, z_2 is exactly the same net but displayed using a different layout, while z_3 is isomorphic to z_1, i.e. identical up to a renaming of places and transitions. In particular, the renaming that has to be applied to places in z_3 is $r_p \triangleq \{p_1 \mapsto p_1, p_2 \mapsto p_3, p_3 \mapsto p_2, p_4 \mapsto p_4\}$, while the transition renaming is $r_t \triangleq \{t_1 \mapsto t_1, t_2 \mapsto t_3, t_3 \mapsto t_2, t_4 \mapsto t_4\}$, finally the required action renaming is $r_a \triangleq \{A \mapsto B, B \mapsto C, D \mapsto A\}$.

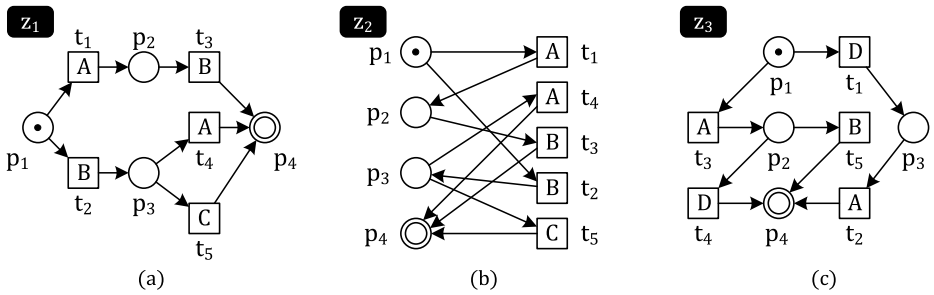


Fig. 5.5. Three equivalent workflow nets with respect their structure: net z_2 is identical to z_1 except for the layout, while net z_3 is isomorphic to z_1 up to a transition and place renaming.

Even if the majority of the available tools allow a free placement of language elements, many unstated conventions are adopted in process design. For instance, models can be expected to be arranged from left to right or from top to bottom, and two sequential transitions are expected to be put one after the other following such orientation, unless they are part of a loop. Moreover, transitions after a choice, like t_4 and t_5 in z_1, are expected to be arranged orthogonally to the model orientation, e.g. from top to bottom if the model is oriented from left to right.

Structure is a key concept for the comprehensibility of graphical models: whenever possible the same logic shall be always represented in the same way, otherwise many advantages of graphical representation would be lost.

Notice that imposing structured rules cannot avoid the creation of bad models or the possibility to obfuscate their meaning: it simply makes such things more difficult. The aim of enforcing a predefined structure is to relief end-users from reasoning about it, so that they can focus only on the process model meaning. The additional overhead induced by following certain structures and conventions is payed-off soon, because a model is primarily built to be read multiple times by different people.

◊

5.5 Modularity

A PML is essentially a tool for tackling the inherent complexity of an existing or desired system; hence, modularity should play a major role in the design of its constructs. Modularity is the property of a system to be decomposed into smaller interrelated parts which can be recombined in different configurations [3]. This notion can be applied to a PML that is said to be modular if its models can be decomposed into smaller reusable components which in turn can be recombined for exploring different design alternatives. Let us stress that the presence of abstractions to decompose a model into smaller parts is necessary to guarantee modularity, but it is not sufficient if such parts cannot be recombined in some way.

A complete formalization of modularity is far from being simple: the efforts needed to make a particular change should be correlated to its global effects; in turn, such efforts depends on language constructs, the run-time support, which steps can be automated and so on. For instance, let us consider the difficulty to change a global variable name with respect to change its type in a large program: changing the name is a time consuming task if performed by hand, but it can be easily automated; conversely, changing the type is trivial because it can be done in only one place, but its consequences should be carefully weighted for not accidently introducing subtle errors. Nevertheless, in characterizing modularity, it can be claimed that any relevant semantic-preserving change should be ideally effortless: this kind of changes is useful for enhancing model comprehensibility and modularity, without compromising the original process behavior. In the following the notion of semantic-preserving decomposition is formalized to prove that an unstructured PML needs AMP (Asynchronous Message Passing) constructs in order to support a minimum level of modularity. In particular, a PML is hierarchically decomposable if any process model can be reduced in size by replacing a selected group of components with a single one containing them and their related state. A generic formalization of decomposability is given by the following definition.

Definition 5.1 (Decomposability). A PML $\mathcal{L}$ is *decomposable* if and only if for any process model $P \in \mathcal{L}$ and for any selected set of components S of P there exists a decomposition $Q, R \in \mathcal{L}$ such that (1) an instance r of type R is a component of Q, (2) R is implemented with the S components, (3) Q does not contain such components, (4) the state shared by the components in S is isolated in the new component R, and (5) Q is equivalent to P:

$$\begin{aligned} &\mathcal{L} \textit{ decomposable} \iff \\ &\quad \forall P \in \mathcal{L} \,.\, \forall S \subseteq \kappa(P) \,.\, \exists Q, R \in \mathcal{L} \,. \\ &\qquad \kappa(R) = S \;\wedge\; \kappa(Q) = (\kappa(P) \setminus S) \cup \{\langle r, R\rangle\} \;\wedge \\ &\qquad \sigma(P, S) \cap \sigma(Q) = \varnothing \;\wedge\; \sigma(P, S) \subseteq \sigma(R) \;\wedge\; Q \sim P \end{aligned} \tag{5.1}$$

where the function $\kappa : \mathcal{L} \to \mathcal{K}$ returns the components of a model, while the function $\sigma : \mathcal{L} \times \mathcal{K} \to \mathcal{Q}$ captures the storage used by a set of components, e.g. the locations assigned to the involved variables. For convenience, $\forall X \in \mathcal{L} \,.\, \sigma(X) \triangleq \sigma(X, \kappa(X))$, while $\mathcal{K}$ and $\mathcal{Q}$ denote components and memory locations related to the language $\mathcal{L}$, respectively. The relation $\sim \subseteq \mathcal{L} \times \mathcal{L}$ represents a notion of equivalence between models. □

Def. 5.1 is generic and needs to be adapted to a particular language by specifying the missing relations: one has to formalize the notion of components $\kappa(X)$ of a model X and what is a component instance $x \in \kappa(X)$. The last one is represented as a pair $x = \langle i, T \rangle$ containing an instance identifier i followed by its type T. The equivalence relation $\sim \subseteq \mathcal{L} \times \mathcal{L}$ between process models Q and P can be a functional equivalence, a trace equivalence or a bisimulation. Moreover, a strict equivalence is often a too strong condition to satisfy and one may only require that Q is able to simulate the behavior of P but not vice versa.

Notice that decomposability is not intended to capture the multifaceted concept of modularity but only a restricted aspect of it. After all, hierarchical decomposition is a necessary but not sufficient condition for enhancing modularity: a large model decomposed into smaller parts can be more comprehensible but potentially more difficult to change. Nevertheless, a PML that does not support decomposability can be hardly considered modular, and its practical use is questionable. To make a concrete example, the following proposition discusses the decomposability of WFNs: the simple modeling language adopted in Chap. 4.

Proposition 5.2 (WFNs Decomposability). The WFNs language is not decomposable: there exist trivial workflow nets that cannot be decomposed into smaller parts without changing their semantics. Since WFNs does not offer a way to store data, i.e. $\forall P \in \mathcal{W} \,.\, \forall S \subseteq \kappa(P) \,.\, \sigma(P, S) = \varnothing$, the inverse of Def. 5.1 can be rewritten in the following way:

$$\begin{aligned} &\exists P \in \mathcal{W} \,.\, \exists S \subseteq \kappa(P) \,.\, \forall Q, R \in \mathcal{W} \,. \\ &\quad \kappa(R) = S \;\wedge\; \kappa(Q) = (\kappa(P) \setminus S) \cup \{\langle r, R \rangle\} \;\Rightarrow\; \neg(Q \sim P) \end{aligned} \tag{5.2}$$

Silent transitions are considered internal constructs, hence workflow nets components become $\forall P \in \mathcal{W} \,.\, \kappa(P) = \{\langle t, a \rangle \mid \langle t, a \rangle \in \mathit{transitions}(P) \wedge a \neq \tau\}$. For sake of simplicity, the equivalence relation $\sim \subseteq \mathcal{W} \times \mathcal{W}$ is defined on trace equivalence because only basic workflow nets with simple observable traces have to be considered: $\forall x, y \in \mathcal{W} \,.\, x \sim y \iff \pi(\mathcal{A} \setminus \{R\}, \mathit{obs}(x)) = \pi(\mathcal{A} \setminus \{R\}, \mathit{obs}(y))$ where $\mathcal{A} \setminus \{R\}$ is the generic set of all visible actions excluding the action adopted as a placeholder for the new component. It is also clear that $\forall \Omega \subseteq \mathcal{A} \setminus \{R\}$ and $\forall x, y \in \mathcal{W},\ x \sim y \Rightarrow \pi(\Omega, \mathit{obs}(x)) = \pi(\Omega, \mathit{obs}(y))$.

Proof. A sound workflow net P that cannot be decomposed for the selected components $S = \{\langle t_4, \mathsf{D} \rangle, \langle t_5, \mathsf{E} \rangle\}$ is shown in Fig. 5.6. Any workflow net $R \in \mathcal{W}$ made of the selected components S has at least the actions $\Sigma = \{\mathsf{D}, \mathsf{E}\}$. Depending on how it is

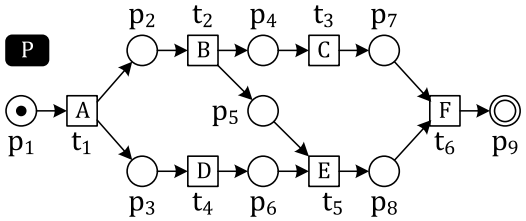


Fig. 5.6. A workflow net that cannot be decomposed.

built, the observable traces of R contain zero or more occurrences of the Σ actions. Since WFNs do not support conditional executions, any observable trace of a workflow net $R \in \mathcal{W}$ has the chance to be executed at least one time. As a consequence, $\pi(\Sigma, obs(R)) = \pi(\Sigma, obs(P)) = \{\textsf{DE}\}$, otherwise $\pi(\Sigma, obs(Q)) \neq \pi(\Sigma, obs(P))$ that makes trivial the conclusion $\neg(Q \sim P)$.

The component $\langle r, R\rangle \in \kappa(Q)$ has to be placed in a path of Q that it is executed at least one time, otherwise $\pi(\Sigma, obs(Q)) = \varnothing$ and $\neg(Q \sim P)$ is again a trivial conclusion. Moreover, $\langle r, R\rangle$ has to be placed in a way that the projected observable traces $\pi(\Sigma \cup \{\textsf{B}\}, obs(P)) = \{\textsf{BDE}, \textsf{DBE}\}$ can be replicated. The component $\langle r, R\rangle$ can be invoked only one time before, after or in parallel with $\langle t_2, \textsf{B}\rangle$, otherwise there are projected traces containing the Σ actions multiple times. If $\exists \sigma \in traces(Q)$ such that $r \xrightarrow{\sigma} t_2$, then $\textsf{DEB} \in \pi(\Sigma \cup \{\textsf{B}\}, obs(Q))$ but $\textsf{DEB} \notin \pi(\Sigma \cup \{\textsf{B}\}, obs(P))$ which implies $\neg(Q \sim P)$. If $\exists \sigma \in traces(Q)$ such that $t_2 \xrightarrow{\sigma} r$ then $\textsf{DBE} \notin \pi(\Sigma \cup \{\textsf{B}\}, obs(Q))$ but $\textsf{DBE} \in \pi(\Sigma \cup \{\textsf{B}\}, obs(P))$ which implies $\neg(Q \sim P)$. The last chance to simulate P using R is placing $\langle r, R\rangle$ in parallel with $\langle t_2, \textsf{B}\rangle$ to obtain both **BDE** and **DBE** but in such case **DEB** is also possible in Q. Therefore, there is no Q having R as a component able to simulate the observable traces of P. □

The same proposition can be also stated for YAWL, even if in this case the presence of variables and conditional executions make the proof more complex.

Proposition 5.3 (YAWL Decomposability). The YAWL language is not decomposable: there exist trivial YAWL models that cannot be decomposed into smaller parts without changing their semantics.

Proof. A YAWL model that cannot be decomposed without changing its semantics is depicted in Fig. 5.7. Except for the use of variables, this model can be easily mapped to the workflow net of Fig. 5.6 following this simple procedure: variables are removed, tasks become transitions, the start place becomes a marked place with one token, the end place becomes an halt place, every arc except for the first and the last one is exploded into two arcs connected by a place. With such mapping in mind, the notions of components $\kappa : \mathcal{W} \to \mathcal{K}$ and equivalence relation $\sim \subseteq \mathcal{W} \times \mathcal{W}$ given for WFNs can be reused for YAWL models.

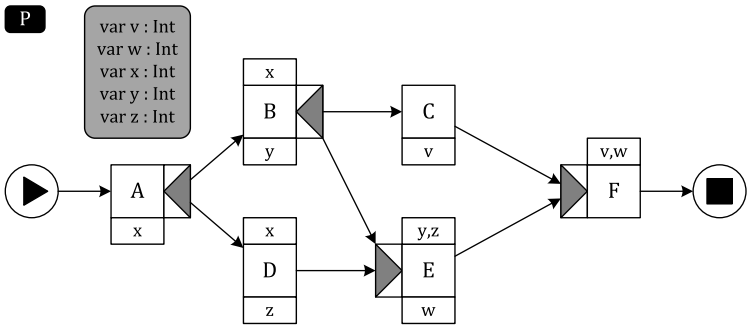


Fig. 5.7. Example of a YAWL process model that cannot be decomposed.

For every YAWL model $Y \in \mathcal{Y}$ and for every component $c \in \kappa(Y)$, $vars(c)$ denotes the set of variables read and written by c, e.g. referring to Fig. 5.7 $vars(B) = \{x, y\}$. For every YAWL model $Y \in \mathcal{Y}$ and a group of components $S \subseteq \kappa(Y)$, $\sigma(Y, S) = vars(S) \setminus vars(\kappa(Y) \setminus S)$. Let us notice that $\sigma(Y) = \sigma(Y, \kappa(Y)) = vars(\kappa(Y)) \setminus vars(\varnothing)$. Considering the set of components $S = \{\langle d, D\rangle, \langle e, E\rangle\}$ selected in the previous proposition, $\sigma(P, S) = vars(S) \setminus vars(\kappa(P) \setminus S) = \{w, x, y, z\} \setminus \{v, w, x, y\} = \{z\}$.

Variable z is only used by D and E for communication purpose, hence it can be considered local to them and needs to be isolated into the new component R. Since R is stateless, E must run after D in the same invocation, otherwise the value of the local variable z will be lost: for isolating z, the components D and E have to be executed in the same context. Furthermore, R cannot run before B because it needs y. As a result, no process Q based on R can have an observable trace containing **DBE** like P: it follows $\neg(Q \sim P)$. □

The isolation of variables used for local communication among components is crucial for the notion of decomposability. If it is removed, it is not difficult to find a decomposition for P, as shown in Fig. 5.8 and Fig. 5.9.

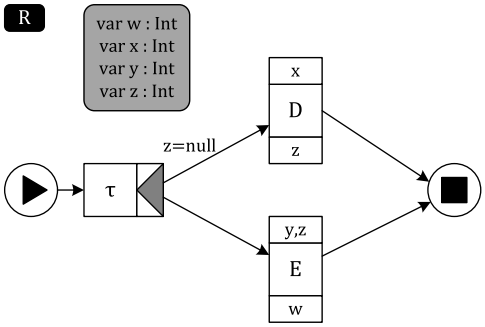


Fig. 5.8. R can run one of the contained components depending on the passed arguments. The conditional execution is based on the variable z.

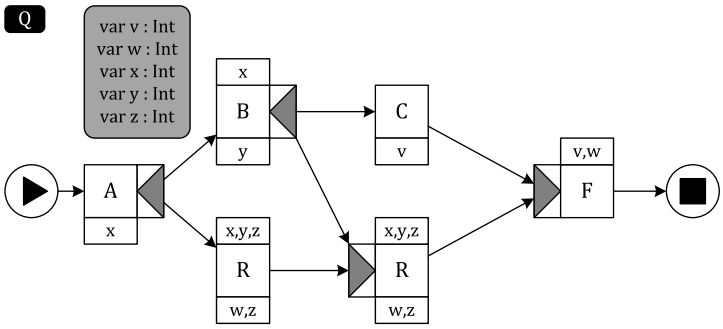


Fig. 5.9. A decomposition Q/R or the process P in Fig. 5.7 that does not respect the state isolation principle.

5.6 Process Design in NestFlow

NestFlow is a novel structured graphical PML conceived to explore a particular language design solution, in which structured control-flow constructs are tightly-coupled with asynchronous message passing for enhancing modularity.

The language is claimed to be *structured* for three main reasons: firstly, control-flow structures can only be expressed by recursively nesting blocks with a single entry and a single exit point and every block has its own distinct representation and meaning, i.e. the same block cannot be used for different purposes. In this way the free-design paradigm is banned: there is no way to freely connect elements with arcs representing control-flow relations; conversely, the design process proceeds by nesting and moving blocks. Secondly, the structure of a model is strictly related to the expressed logics: despite the layout can be adjusted within certain limits, spatial relations among graphical elements are preserved. For example, the components under a choice are always placed after the corresponding graphical element w.r.t the control-flow orientation. Put in a different perspective, the only way to heavily distort the layout of a model without changing its behavior is finding a very different way to express the same logic. Humans are good in recognizing already seen patterns: hence, preserving relations among recurring structures improves model readability. Thirdly, threads of control are confined to specific structures to guarantee that at most one thread is running in a given branch. As a consequence, any concurrent execution is graphically represented by a parallel branch that in turn can be statically declared at design-time or added and removed at run-time using specific commands. The correlation between threads and parallel branches becomes very useful in reasoning about the interaction of concurrent entities.

Complying with predefined structures during the design activity may seem restrictive but the lack of modularity and of layout rules are more serious problems that undermines PMLs usability. The use of structured forms also improves the overall language modularity making a block and its contained parts a good candidate for a decomposition. Furthermore, a structured control-flow eases the integration of AMP constructs that are essential for both modeling non-trivial processes and creating modular reusable components. In NestFlow data-flow constructs are promoted to first-class citizens; conversely, main-stream PMLs do not support AMP at all or even they explicitly forbid it in expressing the logics of a single process, as it happens for instance in YAWL [69] and BPMN [16], respectively. The following example clarifies this different point of view about AMP.

Example 5.4. AMP is essential for enhancing modularity and unraveling complex control-flow relations. For this purpose, message exchanges can be seen as weak control-flow dependencies managed by specific blocks and commands that form the main control-flow logics of a process. In the hypothetical models of Fig. 5.10, components B and D are executed in parallel with C, while E has to wait for its completion before starting. The decision to execute either D or E is taken only after the completion of B. The BPMN model in Fig. 5.10.a runs into a deadlock when the condition $x \geq 0$ evaluates to true, because the thread suspended in j_2 cannot be resumed. In YAWL this situation can be corrected by defining a cancellation region for F that includes the arc between C and E, as in Fig. 5.10.b. In the NestFlow model of Fig. 5.10.c the constructs s and j delimits a parallel

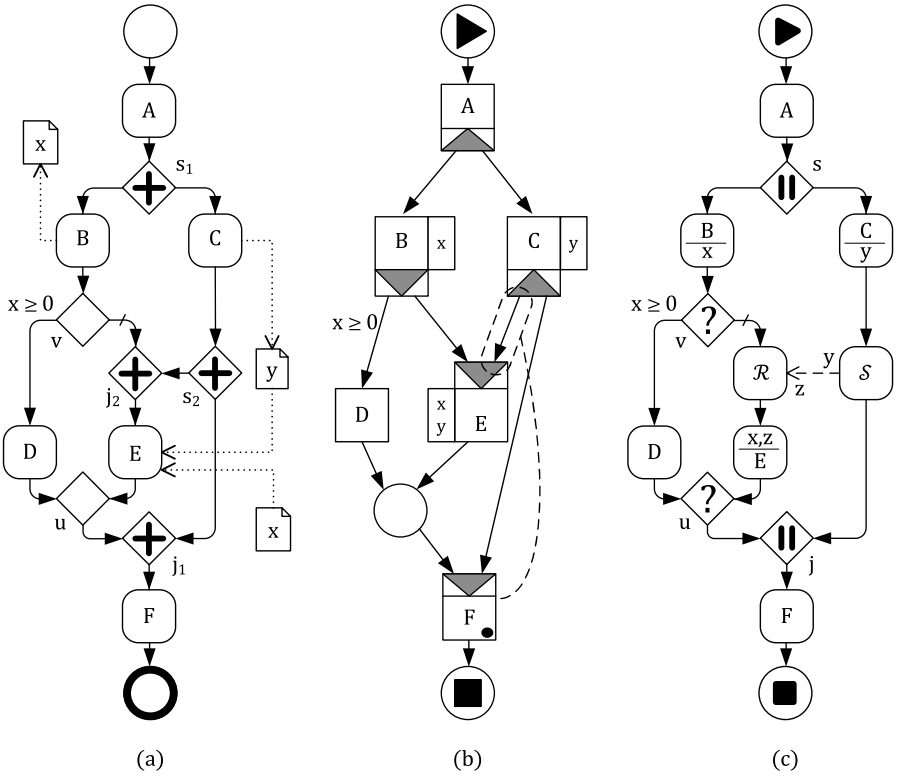


Fig. 5.10. The different roles of messages and threads in control-flow modeling.

block whose left branch contains a choice block between v and u. The dependency between C and E is represented through a data-flow dependency that connects the two parallel branches. It is a reasonable solution because a data-flow dependency between these tasks means that E needs something produced by C for continuing. Whenever $x \geq 0$, the object sent after C is not consumed, but this condition is perfectly acceptable and does not cause any critical fault: data can be retained for the next execution or discarded if they become useless. □

Let us notice that NestFlow has to be considered a proof-of-concept language rather than a fully-fledged system ready to use: its main aim is to show that a structured PML with the discussed features can be effectively built. In its minimality, it cannot be compared to a multifaceted standard like BPMN or a complete working system like YAWL, e.g. there is no direct support for compensation or resource management. Nevertheless, NestFlow offers a wide range of basic constructs and is fairly open for extensions; hence, it is unlikely that its core constructs need to be redesigned to integrate missing features. For example, compensation can be supported with a new notation and eventually implemented over the existing exception handling mechanism. In comparison, the core constructs of CPNs cannot be easily extended without breaking its fundamental principles that make it an ideal language for system verification. This may explain why several PMLs are born after CPNs, despite its unquestionable qualities.

5.7 NestFlow Language Constructs

This section introduces both the bare and core constructs of the NestFlow modeling language as well as how these constructs can be put together to form a complete process model. The section begins with an overview about the constructs and their principal compositional rules, then each one is treated in details in a separate subsection. Core constructs are discussed in a subsequent section that shows how they can be defined as a combination of the existing ones. This approach helps in obtaining a concise introduction, but it is very likely that an actual implementation takes such core constructs as primitive ones. The description of a construct is in general organized in four sections, each one discussing a particular aspect. The *representation* section explains how the construct is graphically represented and encoded in textual form. Notice that the focus remains on graphical representation, while the textual representation is given here only for sake of completeness to show how a process can be encoded facilitating its interpretation or translation. The *syntax* section discusses the grammar rules and the syntactical constraints that state how a construct can be composed with other ones. The *semantics* section describes the construct behavior, namely how it effects the model execution. Finally, when it is present, the *examples* section shows some valid and invalid models that make use of the described construct.

The concrete syntax of a textual programming language can be formally specified through a context-free grammar encoded in one of the many extensions of the Backus-Naur-Form (BNF) meta-language [107]. In similar way, the concrete syntax of the NestFlow bare language is summarized by the graphical BNF-like grammar in Fig. 5.11. In such figure, the dashed rounded boxes enclosing an uppercase letter between angled brackets denote non-terminal symbols. A non-terminal symbol $\langle X \rangle$ can be replaced with any other language element shown in Fig. 5.11 provided that such element is annotated with the same symbol $\langle X \rangle$ and no additional syntactical constraint forbids its nesting. In particular, a BNF grammar rule of the form $\langle X \rangle ::= X_1 | \ldots | X_n$, containing the choice operator $|$, means that $\langle X \rangle$ can be replaced by one block chosen among the available alternatives $\{X_i\}_{i=1}^{n}$.

With reference to Fig. 5.11, the letters P, A, B, C, D, N, T, φ, ψ, ξ, and v enclosed between angled brackets denote non-terminal symbols. The initial symbol of the grammar $\langle P \rangle$ can be replaced with a new process block in which $\langle T \rangle$ is its *name*, $\langle A \rangle$ its *body* and $\langle D \rangle$ some auxiliary declarations. The logic of a process model is stated by expanding its body with the remaining elements of Fig. 5.11 identified by the non-terminal symbols $\langle A \rangle$ and $\langle B \rangle$. An element produced from $\langle A \rangle$, or equivalently from $\langle B \rangle$, can be a *terminal* or a *non-terminal block*. The non-terminal blocks sequence, choice, loop, parallel, concur, try or threshold may have one or more blocks as *children*, while the terminal blocks skip, run, spawn, throw, send, receive or empty cannot be further expanded. A terminal block produced from $\langle C \rangle$ is called *command* and when it is clear from the context a non-terminal block is simply referred as a block.

A block may have several mandatory or optional *inscriptions* that are identifiers or conditions. An *identifier* is a name that uniquely identifies an object in a given scope. Identifiers are classified in *instance*, *type* and *exception identifiers* that replace the non-terminal symbols $\langle N \rangle$, $\langle T \rangle$ and $\langle \xi \rangle$, respectively. The presence

of several identifiers should not scare the reader, because many of them are used for documentation purposes and they can usually be safely omitted. A *condition* is a boolean expression free from side-effects. Every non-terminal symbols $\langle\varphi\rangle$ and $\langle\psi\rangle$ in a block is replaced by a condition that shall be at least a truth value. The condition $\langle\psi\rangle$ placed at the end of a parallel block is called *join condition*, it is distinguished from $\langle\varphi\rangle$ because it is not mandatory and is interpreted in a slightly different way. In particular, when it is not specified, it is assumed to be false. Finally, the non-terminal symbol $\langle v\rangle$ is the identifier of a positive integer variable.

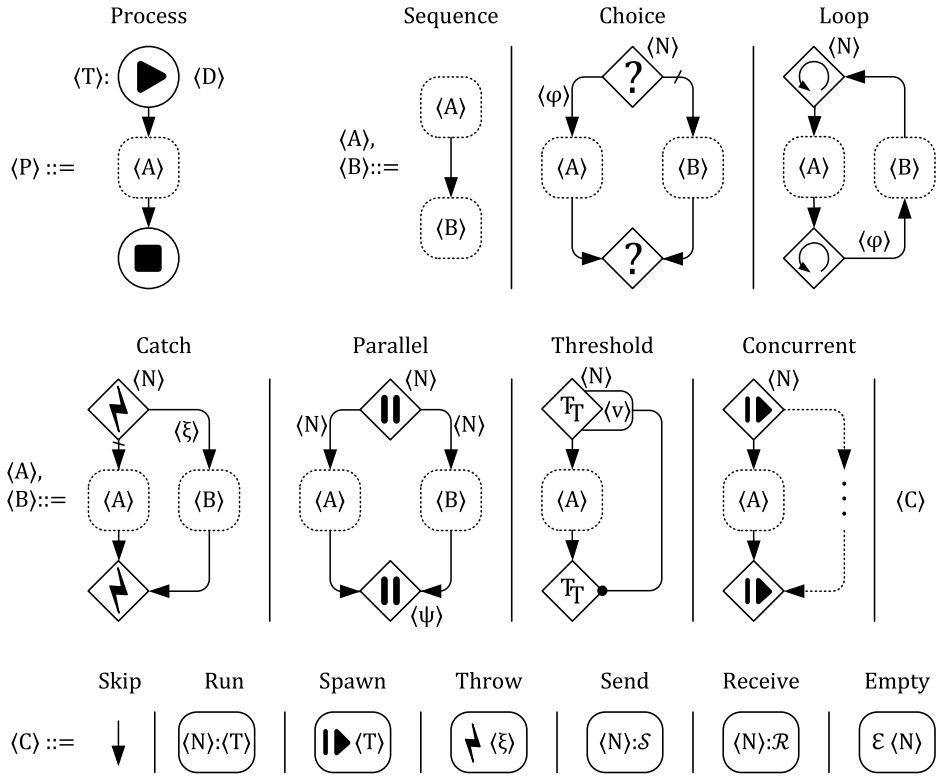


Fig. 5.11. The NestFlow core constructs and its basic syntax.

A model can be oriented left to right or top to bottom. The preferred orientation depends only on the available space and the end-user preferences. An actual implementation should support both orientations making easy to change from one to the other.

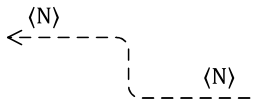


Fig. 5.12. The NestFlow link construct.

The last NestFlow construct is the link exemplified in Fig. 5.12. It is represented as a dashed straight or curved line with an open arrow at one end. The form of a link may vary in length, position and number of straight segments. A link may also *orthogonally* cross several control-flow lines. A link can connect run, send, or receive commands, and its ends are annotated with an identifier, as it will be explained in the following sections.

5.7.1 Basic Elements

The NestFlow definition preserves many aspects of the underlying implementation language, that in this case is Java [108] due to the authors' preference. For example, NestFlow does not introduce a specific notion of exception, it simply grabs the usual class hierarchy of Java exceptions as a starting point. This choice is largely justified by the nature of NestFlow which does not aim to be a standard or fully-fledged system, but try to explore a different PML design solution. After all, it is a common practice in PML design to support certain aspects with one or more existing languages, e.g. the CPNs language takes Standard ML [65] as inscription language, while YAWL adopts XML [109] and XQuery [110] for defining and manipulating data. The drawback of this approach, at least for readers, is the need to known an additional language that at first glance can be seen arbitrarily chosen by the authors. Anyway, this allows one to produce a more compact specification that is also more easier to read.

Java is a widely used general-purpose programming language that falls under the C/C++ [111] language family, at least for its syntax. It can be classified as an imperative, object oriented, memory-managed, strongly-typed language. Java semantics can be learnt from its standard specification [108] and other related documents. Moreover, several research projects in literature deal with the formalization of its semantics, an interested reader can take the work in [112–114] as reference point.

Keywords

The following strings are reserved keywords in NestFlow: hence, they cannot be used as identifiers. Most of them are introduced for encoding graphical elements in textual form. Certain keywords are directly taken from Java, and for avoiding confusion all Java keywords are also reserved.

this	in	out	throws	var	true	false
unbound	extends	process	branch	do	sequence	choice
when	otherwise	loop	until	continue	try	catch
parallel	join	when	threshold	of	concurrent	otherwise
spawn	as	in	throw	send	to	and
receive	from	or	after	empty	run	end

Identifiers

An *identifier* is a non-reserved keyword that starts with a letter or an underscore "_" followed by an optional finite sequence of letters, digits and underscores. An actual implementation can use the same identifiers of the underlying implementation language.

Types

A *type* is a label associated to a language entity in order to uniquely identify its features, e.g. its interface, structure or behavior. A type shall be a valid identifier:

by convention it begins with a capital letter in order to be easily recognized, as happens in Java. An element x of type T is denoted as $x : T$, where x and T are both valid identifiers. Introducing a specific type system is out the scope of this specification: when necessary the Java type system is taken as reference model.

Variables

In NestFlow a *variable* is a name given to a storage location. Every variable has a fixed type. A variable x of type T is declared using the following syntax:

$$\mathsf{var}\ x{:}T; \qquad\qquad \mathsf{const}\ x{:}T;$$

where x is a valid identifier and T an existing type. A variable that never changes its value is said to be *constant* and is declared using the keyword const. Constant initialization can be deferred, as happens in Java with "final", by supporting a single assignment semantics.

When a variable stores an object, it is said to be *bound*, otherwise it is *unbound*. A variable can be initialized using a value c of the right type through the syntax $\mathsf{var}\ x{:}T \leftarrow c$. The specification does not restrict the type of available objects but it often refers to Java immutable objects [108] which include primitive types in wrapped form, such as Integer and String.

The scope of a variable can be statically determined from its declaration. In first analysis, it coincides with a process specification: a variable has to be unique in a process specification and its visibility does not extend outside this boundary. This rule requires to declare all variables at the beginning of a process, but it can be relaxed by allowing variables to be declared inside a block: in this case the variable scope is restricted to the block itself.

Streams

A *stream* is a special element representing an unlimited queue of objects of a predefined type that is used for modeling the flow of objects needed and produced by a process. A stream can be an *input stream* or an *output stream*: objects can be received from an input stream and sent to an output stream. Any stream is uniquely identified in its scope. A stream is declared like a variable but with the keywords in and out that also determine its orientation.

$$\mathsf{in}\ s{:}T; \qquad\qquad \mathsf{out}\ s{:}T;$$

By convention, input stream and output stream identifiers end with the suffixes In and Out, respectively. Greek letters with or without subscripts are used as generic identifiers for streams, e.g. α_{in} can be used as shorthand for alphaIn.

Expressions

In NestFlow a process may be a manual activity directly performed by a human agent, or it may be implemented using a native pre-existing language. As a result, any computable function can be ideally implemented through a native process,

hence fine-grained operations on data can be initially excluded from the bare language specification. In particular, any recursive function $f : A \to B$ can be implemented as a native process F having at least an input stream α_{in} for receiving the arguments and an output stream β_{out} for producing results. In case of simple expressions the process name is replaced by the expression itself that should be read as there exists a native process implementing the specified computation, provided that such computation is feasible.

Conditions

A *condition* is a boolean expression over a set of declared variables. During the execution of a process, a condition is evaluated considering the current value associated to each involved variable. The evaluation result can be `true` or `false`. The specification does not limit the kind of conditions that can be expressed but it is supposed that such expressions are free from side-effects; i.e., conditions do not update variables. Java boolean expressions free from side-effects can be taken as representatives of valid conditions. For example, `(s.length() > 1 && 2*z < 23)` is a valid condition assuming that both `s` and `z` are bounded variables of type `String` and `Integer`, respectively. For convenience, mathematical notation is also adopted to express conditions in a more compact way, hence the previous expression become $(|s| > 1 \wedge 2 \cdot z < 23)$. A generic condition is denoted as $\varphi(\bar{x})$ where $\bar{x}$ is the set of involved variables expressed as a comma separated list of elements; for instance, $(|s| > 1 \wedge 2 \cdot z < 23)$ can be replaced by the generic condition $\varphi(s, z)$ where $\bar{x} = \{s, z\}$ emphasized the involved variables.

Exception

An *exception* in NestFlow is simply an object of a predefined type that inherits from a basic exception type. By convention, exception types end with the suffix `Exception` in order to distinguish them from common types. The usual Java class hierarchy of exceptions can be used in NestFlow.

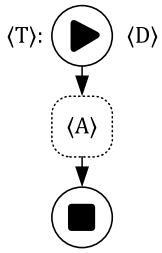
5.7.2 Process Model

In NestFlow a *process model*, or simply a *process* when it is clear from the context, is an executable specification defined combining existing processes with the introduced language constructs. A process model has a unique identifier that is also used as a type to declare new process instances. A *process instance* is a run-time entity that complies with a particular process model. In analogy with object-oriented programming languages, a process is similar to the notion of class and a process instance is the equivalent of an object.

A process model implements at least a process interface. A process instance uses its interface to interact with other instances or with external components through the run-time support. The *interface* of a process T is given by a set of *streams* and a set of *exceptions* that may be raised during its execution. The set of streams is referred to as the *stream interface* of T and it is partitioned in *input streams* and *output streams*. When a process instance $t{:}T$ needs for an object $a{:}A$

to proceed, it declares in its interface an input stream α_{in} of type A. In similar way, if a task t may produce an object $b : B$ relevant for other components, it declares an output stream β_{out} of type B.

Representation – A process model is depicted as in Fig. 5.13.a: the start and end points are marked with a ellipse containing a triangle and a square symbol, respectively. Two straight solid lines ending with an arrow connect these two symbols with the process body $\langle A \rangle$. A process model is surrounded by two main inscriptions $\langle T \rangle$ and $\langle D \rangle$: the former is the process identifier followed by colon, the latter is the process interface declaration containing names and types of input streams, output streams and raised exceptions as shown in Fig. 5.13.b. □



(a)

process $\langle T \rangle$:
  $\langle D \rangle$
  $\langle A \rangle$
end process $\langle T \rangle$;

$\langle D \rangle$ ::=
    in $\langle N \rangle$:$\langle T \rangle$; $\langle D \rangle$
  | out $\langle N \rangle$:$\langle T \rangle$; $\langle D \rangle$
  | throws $\langle T \rangle$; $\langle D \rangle$
  | ε

(b)

Fig. 5.13. (a) A process model declaration, where $\langle T \rangle$ is the process identifier, $\langle A \rangle$ the body of the process and $\langle D \rangle$ is the interface and variable declaration. (b) The textual encoding of a process model declaration.

Syntax – Any combination of blocks and commands can be used to implement the body of a process. An instance of T can refer to one of its own interface stream α through the dot notation this.α, where this is a language keyword. Similarly, a stream α of an invoked process instance $u : U$ is referred to as $u.\alpha$, where u is the instance identifier and α one of its stream. The dot-notation ensures that all streams in a process specification are uniquely identified and this can be left implicit. Any exception not catched by the process logic shall be declared in the process interface hence it can be managed by the caller. □

Semantics – A process can be native or composite depending on how it is implemented. A *native* process is an extension point implemented with a general-purpose programming language. A native process directly interacts with external agents who can be users or other systems; for instance, it can offer a user-interface to support external activities performed by the end-user or it can implement an adapter to drive external programs. A *composite* process is graphically built with the language constructs described in these sections, invoking instances of previously defined processes that in turn can be native or composite. When a process is instantiated there is no difference between native and composite specifications because the interaction occurs through the declared process interfaces. Put in a

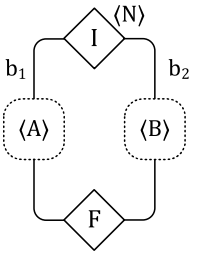
different perspective, a process instance can always be considered an atomic entity decoupled from its actual implementation.

The complete state of a process instance $t:T$ is mainly given by the state of its streams, its variables and the process instances contained in it. In NestFlow a process instance is *stateful* because it retains its state over multiple executions. Any new instance starts from an initial state, which may be modified by sequential executions of the process logic. A stateful component can simulate a stateless component by resetting itself to the initial state at every invocation. □

5.7.3 Control-Flow Blocks and Commands

This section introduces all NestFlow bare constructs used to specify the control-flow logic of a process. The interpretation of a control-flow construct includes the evaluation of zero or more conditions: hence, the emerging behavior is affected by the current state of the process instance. On the contrary, these constructs cannot update variables, because conditions are supposed to be boolean expressions free from side-effects.

Excluding the simpler sequence block, almost all non-terminal blocks share a common graphical style made of two diamond shapes, enclosing a predefined symbol, connected by one or more branches as exemplified in Fig. 5.14.a. The two diamonds I and F represent the initial entry and the final exit point of the control-flow, respectively, while each branch represents a different internal control-flow ramification containing further blocks and commands. Symbols I and F are replaced in each specific block by a particular icon. Branches can have different directions depending on the specific block. The direction is denoted by small solid arrows; when a branch has no particular direction, because it is only used for delimiting the block, it ends with a small bullet. This last kind of branch becomes useful for correctly matching the start and end diamonds of nested blocks. A block can also have an optional identifier $\langle N \rangle$: this may become useful for documentation purposes. A block identifier shall be placed near to the first diamond, but the exact position is not relevant.



(a)

```
block ⟨N⟩:
  branch b1 do
    ⟨A⟩
  branch b2 do
    ⟨B⟩
end block ⟨N⟩;
```

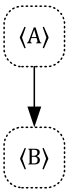
(b)

Fig. 5.14. (a) The prototype of a multi-branch non-terminal block. (b) How it can be encoded in textual form.

As happen for the graphical representation, blocks have a similar textual encoding that is exemplified in Fig. 5.14.b. The encoding starts with the block name followed by a block identifier $\langle N \rangle$ and a colon, then the content of each branch is reported, and finally the encoding ends with the keyword end followed by the block name, its identifier $\langle N \rangle$, and a semicolon. Different keywords are used to delimit branches depending on the block type. Both identifiers $\langle N \rangle$ placed at the beginning and at the end of each block definition can be omitted, but when they are present, they have to be exactly the same. The block name after the end keyword is also optional but useful for documentation purposes.

Sequence Block

Representation – A sequence block is represented without delimiting diamonds by a solid arrow between two arbitrary blocks, as shown in Fig. 5.15.a; the connecting line has a minimum length but it can be stretched whenever necessary to touch the internal components. This may happen when the construct is used in different branches of the same block. The encoding of a sequence block starts with the keyword sequence followed by an optional identifier $\langle N \rangle$ and a mandatory colon, and terminates with the keyword end followed by the optional identifier $\langle N \rangle$ and a mandatory semicolon. □



(a)

sequence $\langle N \rangle$:
  $\langle A \rangle$
  $\langle B \rangle$
end sequence $\langle N \rangle$;

(b)

Fig. 5.15. (a) A sequence is represented by solid line between two blocks ending with an arrow. (b) The encoding of a sequence block.

Syntax – A sequence block can be used in any part of the model to chain two blocks or commands. A chain of blocks of arbitrary length can be obtained by nesting two or more sequence blocks. For sake of simplicity, a sequence block cannot be the first component of another sequence block, hence a chain of sequence blocks can only be obtained expanding the $\langle B \rangle$ non-terminal symbol. □

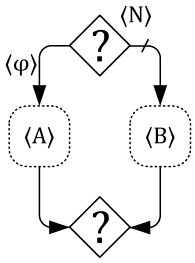
Semantics – A sequence block executes the specified components one after the other. In particular no construct can be executed in the second part if the first one is not completed. □

Choice Block

Representation – The choice block is marked with two diamonds enclosing a question mark, as shown in Fig. 5.16.a. The first branch is annotated with a condition

$\langle\varphi\rangle$, while the second branch is the default one and is marked with an oblique segment. The desired branch order is left to right in a vertical arrangement, and top to bottom in an horizontal one. The order can be reversed if needed considering that the default branch is always marked.

The encoding of a choice block starts with the keyword choice followed by an optional identifier $\langle N\rangle$ and a mandatory colon, and terminates with the keyword end followed by an optional choice keyword, an identifier $\langle N\rangle$ and a mandatory semicolon. Inside this encoding, the declaration of the first branch starts with the keyword when followed by the condition and the do keyword, while the declaration of the second branch starts with the keyword otherwise. □



(a)

choice $\langle N\rangle$:
  when $\langle\varphi\rangle$ do
    $\langle A\rangle$
  otherwise
    $\langle B\rangle$
end choice $\langle N\rangle$;

(b)

Fig. 5.16. (a) A choice block with two branches: the default one is marked with an oblique segment. (b) The encoding of a choice block.

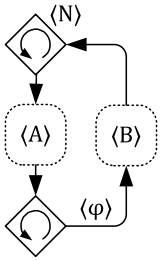
Syntax – A choice block can be used in any part of the model, and each branch can contain any type of block or command. The condition in the first branch is mandatory, as well as the marking of the default branch. □

Semantics – A choice evaluates the condition $\langle\varphi\rangle$ associated to the first branch, if it is satisfied the corresponding branch is executed, otherwise the default one is chosen. Its semantics resembles the usual if-then-else construct of a general-purpose programming language. The name choice has been used instead of if because it is more consistent with its generalized version supporting multiple branches. □

Loop Block

Representation – The loop block is marked with two diamonds enclosing a semicircle with an arrow at one end, as depicted in Fig. 5.17.a. The block contains a forward branch $\langle A\rangle$, and a backward branch $\langle B\rangle$, the latter annotated with a loop condition $\langle\varphi\rangle$. The forward branch is aligned with the diamond symbols, while the backward branch is preferably putted on the right, but this is not mandatory.

The encoding of a loop block starts with the keyword loop followed by an optional identifier $\langle N\rangle$ and a mandatory colon; while it terminates with the keyword end followed by the an optional loop keyword, an identifier $\langle N\rangle$ and a mandatory semicolon. Inside this encoding the specification of the two branches is separated



(a)

loop ⟨N⟩:
  ⟨A⟩
  until ⟨φ⟩ continue
  ⟨B⟩
end loop ⟨N⟩;

(b)

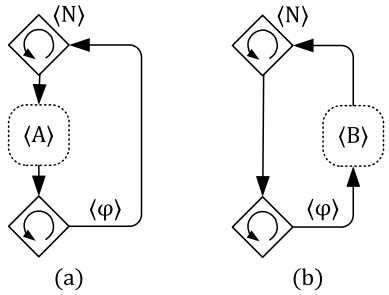
Fig. 5.17. (a) A loop block contains two branches, the second one is executed after the first one only if the condition ⟨φ⟩ evaluates to true. (b) The encoding of a loop block.

by the condition declaration, which is enclosed between keywords until and continue. □

Syntax – A loop block can be used in any part of the model, and each branch can contain any type of block or command. The specification of the loop condition on the backward branch is mandatory. □

Semantics – A loop block executes the content of ⟨A⟩ at least one time, then it executes ⟨B⟩ followed by ⟨A⟩ zero or more times depending on the evaluation of the condition ⟨φ⟩. □

Example – A repeat-until loop can be obtained by replacing ⟨B⟩ with a skip command, as in Fig. 5.18.b, where the skip arrow is fused with the loop backward arrow to form a unique arrow connecting the exit point with the entry one. Similarly, a while-do loop can be obtained by substituting ⟨A⟩ with a skip command, as depicted in Fig. 5.18.b. □



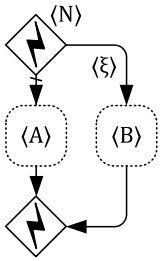
(a) (b)

Fig. 5.18. A while-do loop in NestFlow.

Try-Catch Block

Representation – The try block is enclosed between two diamonds containing a thunderbolt symbol, as shown in Fig. 5.19.a. This block has two branches, the default one is aligned with the diamonds and annotated with a solid oblique segment, while the other one is preferably placed on the right and annotated with an exception identifier $\langle \xi \rangle$. The exception branch can be positioned on the left if this increases the overall model readability.

The encoding of a try block starts with the keyword try followed by an optional identifier $\langle N \rangle$ and a mandatory colon, and it ends with the keyword end followed by an optional try keyword, an identifier $\langle N \rangle$ and a mandatory semicolon. Inside this encoding the specification of the two branches is separated by the exception declaration, which is contained between the keywords catch and do. □



(a)

try $\langle N \rangle$:
  $\langle A \rangle$
  catch $\langle \xi \rangle$ do
  $\langle B \rangle$
end try $\langle N \rangle$;

(b)

Fig. 5.19. (a) A try block with two branches: the first one is marked with an oblique segment and represents the default execution, while the second one is annotated with an exception identifier $\langle \xi \rangle$. (b) The encoding of a try block.

Syntax – A try block can be used in any part of the model, and each branch can contain any type of block or command. The exception identifier on the second branch is mandatory. □

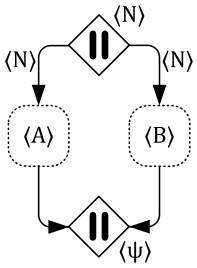
Semantics – A try block executes the first branch $\langle A \rangle$ and if an exception of type $\langle \xi \rangle$ is raised inside it, the execution is interrupted and resumed from the other branch $\langle B \rangle$. Exceptions may be raised by blocks, process instances, or an explicit throw command inside the branch $\langle A \rangle$. When an exception of the specified type is raised, all blocks containing it are recursively reverted until a proper try block is reached. If no try block can manage the given exception, the entire process is reverted and the exception propagated to the caller, similarly to what happens in modern programming languages like Java [108]. □

Parallel Block

Representation – A parallel block is graphically enclosed within two diamonds containing two parallel segments, as shown in Fig. 5.20.a. These two diamonds identify the entry and the exit point of the block. The exit point can be annotated

with an optional join condition $\langle\psi\rangle$. The block has two branches that can be marked with a unique identifier $\langle N\rangle$. These identifiers can be useful in advanced exception handling.

The encoding of a parallel block starts with the keyword parallel followed by an optional identifier $\langle N\rangle$ and a mandatory colon, and it finishes with the keyword end followed by the optional parallel keyword, an identifier $\langle N\rangle$ and a mandatory semicolon. Inside this encoding the declaration of each branch is preceded by the keyword branch enriched with the optional branch identifier $\langle N\rangle$ and followed by a do. When necessary, the join condition $\langle\psi\rangle$ is specified using the keyword join when followed by a semicolon. □



(a)

parallel $\langle N\rangle$:
  branch $\langle N\rangle$ do
    $\langle A\rangle$
  branch $\langle N\rangle$ do
    $\langle B\rangle$
  join when $\langle\psi\rangle$;
end parallel $\langle N\rangle$;

(b)

Fig. 5.20. (a) A parallel block with two branches and an optional join condition $\langle\psi\rangle$ at the end. (b) The encoding of a parallel block.

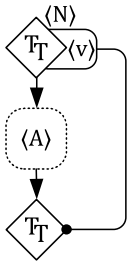
Syntax – A parallel block can be used in any part of the model, and each branch can contain any kind of block or command. The partial join condition is optional. Parallel branches cannot share variables, they can communicate only through message passing. □

Semantics – A parallel block executes the two specified branches in parallel each one with its own thread of control. The condition $\langle\psi\rangle$ at the end of the parallel block can be used to define a generalized partial join. This condition is evaluated whenever possible on the available variables every time a parallel branch terminates. A variable x used in $\langle\psi\rangle$ is available if it is not involved in a running parallel branch, otherwise it is considered unknown and so the part of $\langle\psi\rangle$ concerning x. When $\langle\psi\rangle$ is true, the remaining running branches are cancelled by raising an interrupting exception. In any case, a parallel block is left if and only if all threads have been completed or reverted. Parallel branches cannot share variables, but they can communicate through message passing. This constraint can be relaxed for read-only variables: if a variable is used only as a right value in assignments, it can be safely accessed in a shared way. □

Threshold Block

Representation – A threshold block is graphically depicted within two diamonds marked with a double T symbol, resembling the initials letters of the words thread threshold, and a variable v representing the maximum number of threads that can execute concurrently inside the block, as depicted in Fig. 5.21.a. This block has only one body $\langle A \rangle$ and the two diamonds are connected with a line terminating with a solid bullet. This line is used to mark the block scope and is particularly useful for determining the correct nesting of several threshold blocks when they are used in sequence or one inside the other .

The encoding of a threshold block starts with the keyword threshold followed by an optional identifier, the keyword of, the number of available threads $\langle v \rangle$ and a colon, and it ends with the keyword end, followed by the optional block name, an identifier $\langle N \rangle$, and a mandatory semicolon. □



(a)

threshold $\langle N \rangle$ of $\langle v \rangle$:
  $\langle A \rangle$
end threshold $\langle N \rangle$;

(b)

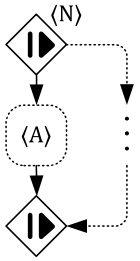
Fig. 5.21. (a) A threshold block denoting the maximum number of threads tha can concurrently run inside its body. (b) The encoding of a threshold block.

Syntax – A threshold block can be placed in any part of the model and its body can contain any kind of block or command. The identifier $\langle v \rangle$ has to refer to a variable containing a positive integer. □

Semantics – A threshold block limits to v the number of threads that can execute concurrently inside its body. The number of existing threads remains the same and corresponds to the number of declared parallel branches, but if the number n of existing threads is greater than v, $n - v$ threads will be suspended until one of the first v threads has completed its execution or it enters into a waiting state. □

Concurrent Block

Representation – A concur block is graphically enclosed within two diamond symbols containing a triangle aligned with a vertical bar, as depicted in Fig. 5.22.a. The block has one solid branch $\langle A \rangle$ representing the default execution, and one or more dashed branches representing dynamic parallel branches, that may be created at run-time with a spawn command inside its body. A dashed branch will be



(a)

concurrent ⟨N⟩:
  ⟨A⟩
end concurrent ⟨N⟩;

(b)

Fig. 5.22. (a) A concur block with a main branch ⟨A⟩ and a dashed branch representing where to place a dynamically created branch. (b) The encoding of a concur block.

dynamically created for each spawned process instance and removed when such instance terminates.

The encoding of a concur block starts with the keyword concurrent followed by an optional identifier and a mandatory colon, and it ends with the keyword end followed by the optional block name, an identifier, and a mandatory semicolon. □

Syntax – A concur block can be used in any part of the model and its solid branch can contain any kind of block or command. Its body has to contain at least a spawn command referring to it, that can potentially create a new process instance at run-time when executed. □

Semantics – A concur block is the scope of a spawn command and its behavior does not differ substantially from a parallel block. It initially executes ⟨A⟩ but one or more parallel branches can be added at run-time using a spawn command; all threads join before exiting the block. A spawn/concur pair provides a graphical representation of dynamically created instances based on run-time model variation. PMLs usually do not offer any representation of this dynamic behavior and concurrent instances are often left implicit. □

Skip

Representation – A skip command is graphically represented as a simple solid arrow of variable length. When the skip arrow connects two control-flow lines, they can be merged into a unique line with a single arrow. The encoding of a skip command is simply the keyword skip followed by a semicolon. When it is clear that a branch is empty, it can be omitted. □

Fig. 5.23. A skip command.

Syntax – A skip can be used in any part of the model without limitation. □

Semantics – A skip command does not produce any effect. It is is useful for obtaining specific control-flow structures from generic ones; for instance,

the usual repeat-until and while-do loops can be obtained replacing the first or the second branch of a loop block with a skip, respectively. □

Run

Representation – A run command has no special symbol, it is graphically denoted as in Fig. 5.24.a with a rounded box containing the process name and its type in the usual notation $\langle N \rangle : \langle T \rangle$. The instance identifier $\langle N \rangle$ can be omitted and used only when strictly necessary or for documentation purposes. The encoding of a run command starts with the keyword run followed by the type identifier $\langle T \rangle$, the optional keyword as, the instance identifier $\langle N \rangle$ and a mandatory colon. It terminates with the keyword end, followed by the optional keyword run, the instance identifier and a mandatory semicolon. Inside this encoding the mappings between the input streams of the current process instance and the output streams of another process instance are given using the keyword require, while the mappings for the output streams begin with the keyword offer. Each input mapping is composed of the stream identifier, the keyword from and the output stream of another process specified using the dot notation. The output stream mappings are given in a similar way using the keyword to in place of the from one. □

(a)

```
run ⟨T⟩ as ⟨N⟩
   require ⟨N⟩ from ⟨N⟩.⟨N⟩
      (, ⟨N⟩ from ⟨N⟩.⟨N⟩)*
   offer ⟨N⟩ to ⟨N⟩.⟨N⟩
      (, ⟨N⟩ from ⟨N⟩.⟨N⟩)*
end run ⟨N⟩;
```

(b)

Fig. 5.24. (a) A run command regarding a process instance with identifier $\langle N \rangle$ and type $\langle T \rangle$. (b) The encoding of a run command.

Syntax – There is no limit in the use of a run command, but the invocation of a particular process instance can occur only in one place of a process specification. This restriction can be relaxed but has positive effects in process visualization. □

Semantics – A run command executes a process instance, synchronously suspending the current thread of control until the instance is completed or reverted by an internal exception. Exceptions can be caused by internal or external signals. □

Spawn

Representation – A spawn command is graphically represented with a rounded box containing the same symbol of the concur block and the type $\langle T \rangle$ of the process to be instantiated.

The encoding of spawn command starts with the keyword spawn followed by the process type $\langle T \rangle$, the keyword as together with the expression $\langle e \rangle$ indicating

where to store the new process instance identifier, the optional keyword in followed by a concur block identifier $\langle N \rangle$ and a mandatory semicolon. When the keyword in is omitted, the new process instance is added to the innermost concur block. □

(a)

spawn $\langle T \rangle$ as $\langle e \rangle$ in $\langle N \rangle$;

(b)

Fig. 5.25. (a) The spawn command creates a new process instance of type T returning a new identifier that can be stored into a variable. (b) The encoding of a spawn command.

Syntax – A spawn command can be executed only inside concur block. It the block identifier $\langle N \rangle$ is present, the spawn command should be enclosed in a concur block having such identifier. For sake of simplicity, the expression $\langle e \rangle$ can be only a variable name or an array location. □

Semantics – A spawn command creates a new process instance of the specified type. Such instance is immediately executed into a new parallel branch dynamically added to the inner concur block containing the command, or to the concur block annotated with the specified identifier $\langle N \rangle$, provided that the spawn command is contained in its body. A spawn command returns a unique process instance identifier that can be used for communication purposes.

The idea behind the concur/spawn pair is to give an explicit representation of dynamically created instances; in process visualization, the explicit representation is obtained by expanding and shrinking concur blocks at run-time. In unstructured modeling languages with token-based semantics, the creation of new task instances is usually implicit and it is hard to find a satisfactory run-time representation. □

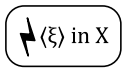
Throw

Representation – A throw command is graphically depicted as in Fig. 5.26.a with a rounded box containing a thunderbolt symbol followed by an exception identifier $\langle \xi \rangle$. A throw command inside a parallel block raising an InterruptException can also be annotated with a list of branch identifiers $\{\langle N \rangle \dots \langle N \rangle\}$ specifying which threads will be interrupted, as in Fig. 5.26.b. Alternatively, the form $\neg\{\langle N \rangle\}$ can be used to specify that the exception will be raised for all branches of the parallel block except for the one with the given identifiers.

The command encoding starts with the keyword throw followed by the exception identifier $\langle \xi \rangle$, the optional specification of parallel branches preceded by the keyword in, and a mandatory semicolon.
Syntax – A throw command can be used only inside a proper try block or inside a process that declares to raise the corresponding exception in its interface. □

Semantics – A throw command raises an exception of the specified type, recursively reverting all blocks that contain it until a proper try block is reached. If no try block is able to handle the exception, it is re-thrown outside the process instance

(a)



(b)

throw ⟨ξ⟩ in {⟨N⟩ . . . ⟨N⟩};

(c)

Fig. 5.26. (a) A throw command which raises an exception of type ⟨ξ⟩. (b) The extended version of the throw command where X is a set of parallel branch identifiers. (c) The encoding of a throw command.

which enters into a failure state. An unhandled exception exits from all nested process instances and it is handled by the run-time support, causing an abnormal termination. A throw command inside a parallel block annotated with one or more branch identifiers and raising an InterruptException will affect only the specified branches, while the other ones normally continue their execution if possible. □

Send

Representation – A send command is graphically represented with a rounded box containing an instance identifer and the $\mathcal{S}$ symbol separated by a colon. The instance identifier is optional and when it is not specified, also the colon is omitted. This notation mostly resembles the one used for a run command, because a send can be actually considered a special atomic process instance of type $\mathcal{S}$.

The encoding of a send command starts with the keyword send followed by an optional instance identifier and a mandatory colon, and it terminates with the keyword end followed by the optional send keyword, an identifier and a mandatory semicolon. Inside this block a mapping between variables and stream identifiers is provided. In particular, the mapping starts with the variable identifier followed by the keyword to and a stream identifier. The stream identifier has the form $\langle N \rangle.\langle S \rangle$ where $\langle N \rangle$ is a process instance identifier and $\langle S \rangle$ is the identifier of a stream belonging to the specified process instance. □

(a)

send ⟨N⟩:
  ⟨N⟩ to ⟨N⟩.⟨S⟩
  (and ⟨N⟩ to ⟨N⟩.⟨S⟩)*
end send ⟨N⟩;

(b)

Fig. 5.27. (a) A send command is graphically represented as a rounded box containing the $\mathcal{S}$ symbol. (b) The encoding of a send command.

Syntax – A send can be used in any part of the model without limitation. The involved streams have to be connected with a link annotated with the given variable. A send command can only store the value of a variable into an output stream w.r.t. the current process instance, which in turn can be an interface output or input stream of an invoked process instance. □

Semantics – A send command inserts the value of one or more variables into one or more corresponding streams. The sending is asynchronous and the execution continues with the next block without waiting. □

Receive

Representation – A receive command is graphically represented as in Fig. 5.28.a with a rounded box containing an instance identifier and an $\mathcal{R}$ symbol separated by a colon. The instance identifier $\langle N \rangle$ is optional and when it is omitted, the colon is also omitted. Similarly to the send, the graphical representation of a receive is a rounded box containing the $\mathcal{R}$ symbol.

The encoding of a receive command starts with the keyword receive followed by the optional instance identifier and a mandatory colon. It terminates with the keyword end followed by the optional receive keyword, an instance identifier and a mandatory semicolon. Inside this block one or more mappings between a variable and a stream identifier are provided separated by the or keyword. In particular, the mapping reports the variable identifier $\langle N \rangle$, followed by the keyword from and the stream identifier $\langle S \rangle$ related to the corresponding process identifier $\langle N \rangle$ through the dot notation. If more than one mapping is specified, they are connected with the keyword or. The timeout associated to a receive command is encoded by specifying the keyword after followed by the timeout duration $\langle \theta \rangle$ and the keyword do, then the encoding of the corresponding throw command with a TimeoutException is reported, followed by the keyword end and a mandatory semicolon. □

(a)

```
receive ⟨N⟩:
  ⟨N⟩ from ⟨N⟩.⟨S⟩
  (or ⟨N⟩ from ⟨N⟩.⟨S⟩)*
  after ⟨θ⟩ do
    throw ⟨ξ⟩
  end;
end receive ⟨N⟩;
```

(b)

Fig. 5.28. (a) A receive command is graphically represented as a rounded box containing the $\mathcal{R}$ symbol. (b) The encoding of a receive command.

Syntax – A receive command can be used in any part of the model without restrictions. The involved streams have to be connected with a link annotated with the given variable. A receive command can only put into a variable the value contained in an input stream w.r.t. the current process instance, which in turn can be an interface input or output stream of an invoked process instance. □

Semantics – A receive stores into one variable an object extracted from one of the available input streams. The receive temporally suspends the current thread of control until an object arrives or a timeout θ raises an exception. A multiple receive

stores the first arrived object from a stream α_{in}^{i} into the corresponding variable x_i, resets the others to unbound and continues the execution: this behavior will be called *or-receive*. □

Empty

Representation – An empty command is represented with a rounded box containing the $\mathcal{E}$ symbol followed by a stream identifier $\langle N \rangle$, as in Fig. 5.29.a. The stream identifier can be specified starting from a process instance identifier using the dot notation. The encoding of a empty command is simply composed of the keyword empty followed by the stream identifier and a semicolon. □

(a)

empty $\langle N \rangle.\langle S \rangle$;

(b)

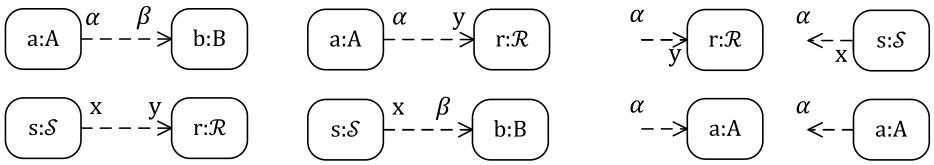
Fig. 5.29. (a) A empty command related to a stream $\langle N \rangle$. (b) The encoding of a empty command emphasizing the dot notation for streams.

Syntax – An empty command can be used in any part of the model. It is only required that the specified stream identifier exists and refers to an input stream for the current process. □

Semantics – The empty command removes all the objects stored in a stream. If the stream is already empty, it does not wait. □

Links

The flow of objects among tasks is represented through *links* graphically denoted with dashed arrows, as in Fig. 5.30. Links can be distinguished in external and internal links depicted in Fig. 5.30.a and Fig. 5.30.b, respectively. Links are annotated with stream and variable identifies depending on the source and destination component. In particular, links on send and receive commands are annotated with variable identifiers, while links on process instances are annotated with stream identifiers. The different combinations are given in Fig. 5.30.



(a) (b)

Fig. 5.30. Different combinations of link notations. (a) Connections between internal instances in the same process specification. (b) Notation used for external streams.

5.8 NestFlow Core Constructs

In this section the NestFlow bare language is enriched with a new set of constructs that are not essential for improving language expressiveness but very useful for obtaining more compact specifications. These new constructs can be easily encoded as a combination of the existing ones reducing formalization efforts. However, the resulting core language is a better starting point for an actual implementation, because bare constructs are over simplified for such purpose.

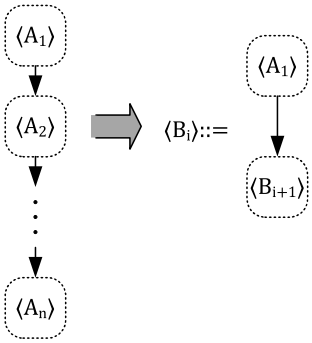


Fig. 5.31. Generic sequence block of n components.

The NestFlow core constructs with their mapping semantics are summarized from Fig. 5.31 to Fig. 5.39, where $n \in \mathbb{N} \setminus \{1\}$ and for every $i \in [1, n]$, $\langle A_i \rangle$ and $\langle B_i \rangle$ are non-terminal symbols and $\langle B_i \rangle = \langle A_i \rangle$ when $i = n$. Many core language constructs are a generalization of the existing ones obtained by nesting multiple times the same structure. By convention, the nesting is always performed on the last branch or component of a block, unless otherwise specified, and $\langle N_i \rangle = \langle N \rangle$ if $i = 1$, or ε otherwise.

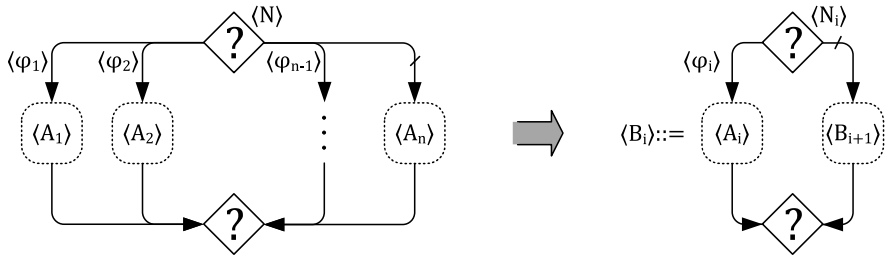


Fig. 5.32. Generic choice block with n guarded branches.

A generic sequence block of n components can be obtained by nesting $n-1$ basic sequence blocks, as shown in Fig. 5.31. A generic choice block with n guarded branches is obtained in similar way by nesting $n-1$ basic choice blocks, as depicted in Fig. 5.32. This construction makes clear in which order the conditions are evaluated. A while-loop structure can be obtained using a loop block and a skip command, as depicted in Fig. 5.33.b; for not placing both the condition and the body in the backward branch, this construction is re-

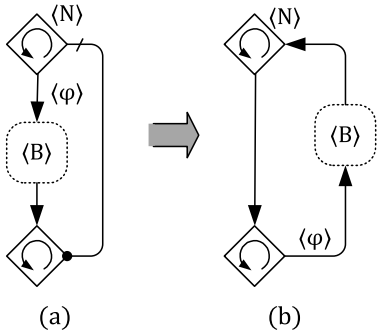


Fig. 5.33. While-loop structure with a condition on the forward branch.

laxed as shown in Fig. 5.33.a, where the right branch has no direction because it is used for exiting the loop when the condition is false and for performing another iteration. A generic try block with n branches able to manage $n-1$ not necessarily distinct exceptions can be obtained by nesting $n-1$ basic try blocks, as illustrated in Fig. 5.34. The folding is performed on the default branch, hence

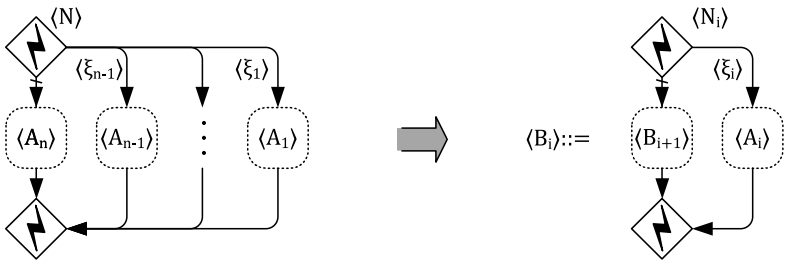


Fig. 5.34. Generic try block capturing n distinct exceptions.

exceptions are evaluated from the nearest to the farthest, namely from left to right or top to bottom depending on the model orientation. Let us notice that exceptions raised in a catch branch can be in turn catched by the remaining branches. A generic parallel block of n parallel branches follows the same construction logic repeating the basic parallel block $n-1$ times, as in Fig. 5.35.

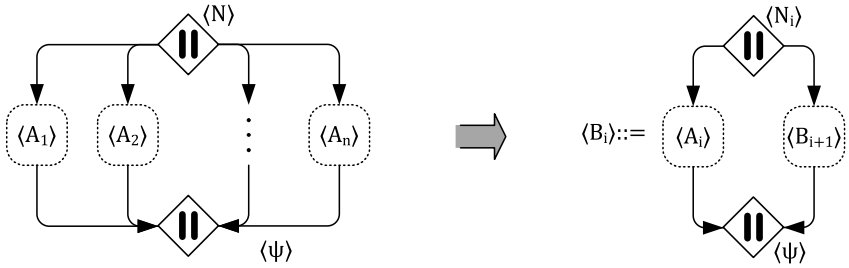


Fig. 5.35. Generic parallel block with n branches.

The concur block can be made more compact by stacking process instances having the same type, as done in Fig. 5.36, in such way there is at most a parallel branch for each process type spawned inside its scope. A small counter x_i near each stack can be added to visualize the actual number of active instances in the branch. A block implementing a structured synchronized merge [29] of n branches can be obtained adding a basic choice block to each branch of a generic parallel

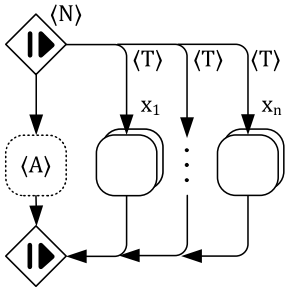


Fig. 5.36. Alternative representation of a concur block with a dashed branch for each spawned process type.

block, where the condition $\langle\varphi_n\rangle$ of the last choice becomes $\langle\varphi_n\rangle = \bigwedge_{i=1}^{n-1} \neg\langle\varphi_i\rangle$ for implementing the default branch. Fig. 5.37 shows how to obtain a structured synchronizing merge block starting from basic constructs.

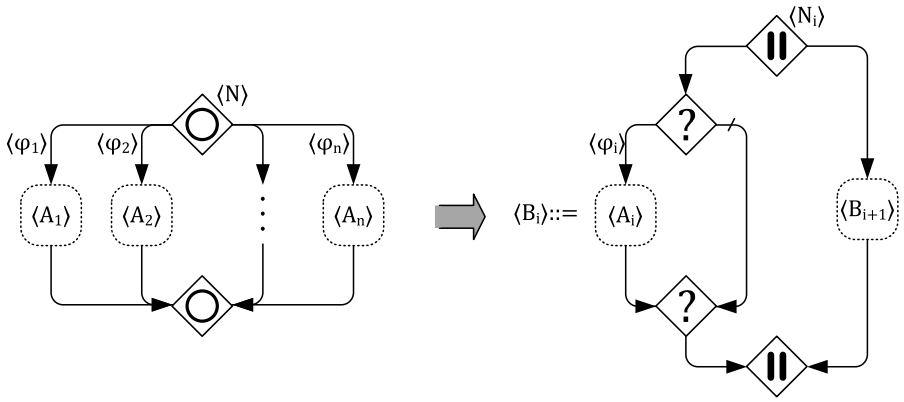


Fig. 5.37. A new block implementing a structured synchronized merge depicted with a different symbol.

A sequence of receive commands can be grouped into a unique receive called *and-receive* which waits for an object from each connected stream before proceeding. An *and-receive* is depicted as in Fig. 5.38 by adding a small circled ∧ symbol to the default receive representation. AMP is the only abstraction available in NestFlow bare language to exchange data with the environment and the internal process instances. This mechanism has been adopted for its simplicity and expressiveness: it can be used to model the communication between concurrent entities and it can easily simulate parameter passing in a sequential context. Parameter passing in the NestFlow core language is denoted as in the process instance $t{:}T$ on the left of Fig. 5.39.a, where input variables $\bar{x} = \langle x\rangle_{i=1}^{n}$ are annotated on the side touched by the incoming arc while output variables $\bar{y} = \langle y\rangle_{i=1}^{n}$ are annotated on the opposite one. Assuming that input and output streams are ordered in some way, e.g. taking the declaration order, each input argument and each return value can be mapped to a stream having a compatible type.

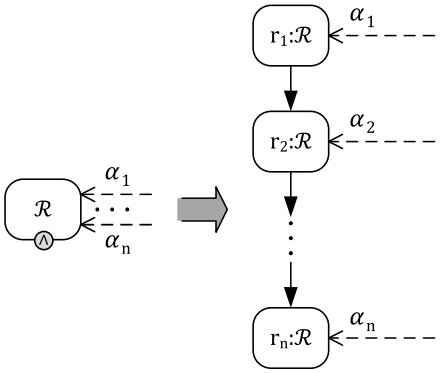


Fig. 5.38. An *and-receive* command.

The translation in terms of basic send and receive commands is exemplified in Fig. 5.39.b, where the superscripts of input streams $\langle\alpha_{in}^{i}\rangle_{i=1}^{n}$ and output streams $\langle\beta_{in}^{j}\rangle_{j=1}^{m}$ are used to emphasize the order among streams.

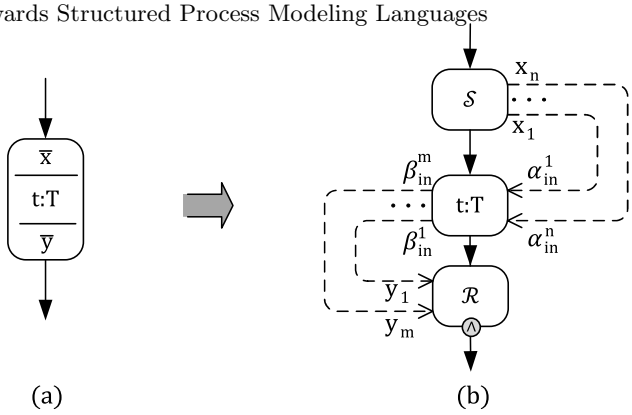


Fig. 5.39. Implementation of parameter passing mechanism using AMP.

Data-flow constructs can be decorated with icons in order to adapt the language to a particular application domain; e.g., they can be depicted as in Fig. 5.40.

Fig. 5.40. Decoration of data-flow constructs with icons resembling messages.

◊

5.9 NestFlow Formal Semantics

This section formalizes the NestFlow bare language: initially the mathematical structures needed to represent a process model are introduced; then the most important well-formedness properties are discussed, showing how they can be statically checked. These properties determine what is a valid model. Finally, the section specifies how valid models are interpreted.

NestFlow core constructs are defined in terms of basic ones; hence, they do not need a wide treatment. Core constructs can be described by introducing new mathematical structures and a translation function towards the bare language. Alternatively, the formal semantics of the core constructs can be directly given making the verification of well-formedness properties and the definition of the interpreter a slightly more complex. In both cases, core construct formalization does not add very new information.

5.9.1 NestFlow Formal Syntax

This section defines the mathematical structures needed to formally represent a process model starting from some basic definitions. In the following, valid identifiers are considered grouped into three disjoint sets $\mathcal{I}, \mathcal{T}, \mathcal{X} \subseteq \mathcal{U}$, such that $\mathcal{I}$ are variable and instance identifiers, $\mathcal{T}$ contains the identifiers used for types, while $\mathcal{X}$ contains the types used for exceptions. It is assumed that the symbol ς does not belong to $\mathcal{I} \cup \mathcal{T} \cup \mathcal{X}$ because it is used to compactly denote the current process instance. No other information or constraint about their definition is given here, since they are considered implementation details. By convention the symbol $t \in \mathcal{I}$ is commonly used for denoting a generic instance identifier, $T \in \mathcal{T}$ a generic type, $\xi \in \mathcal{X}$ a generic exception, while $\mathsf{S}, \mathsf{R} \in \mathcal{T}$ are special symbols used for specifying the type of a send and a receive command, respectively.

A *variable* carries a type with it, hence it should be generally understood as a pair $\langle t, T \rangle$ made of an identifier $t \in \mathcal{I}$ and a type $T \in \mathcal{T}$. The set of all variables is denoted as $\mathcal{V} \subseteq \mathcal{I} \times \mathcal{T}$, while functions $id : \mathcal{V} \to \mathcal{I}$ and $type : \mathcal{V} \to \mathcal{T}$ are used to recover the two components. When it is clear from the context, the symbol used to denote a variable $v \in \mathcal{V}$ is also used in place of the identifier $id(v)$, and when the type T of a variable v is relevant for the discussion, the notation $v{:}T$ is used to declare its type. Similarly to what done for variables, the set of all *streams* with their type are denoted as $\mathcal{S} \subseteq \mathcal{I} \times \mathcal{T}$, while the set of all *components* with their type are denoted as $\mathcal{K} \subseteq \mathcal{I} \times \mathcal{T}$. Streams and components can be understood as special variables since they are placed in the store, but for convenience it is assumed $\mathcal{S} \cap \mathcal{V} = \varnothing$ and $\mathcal{K} \cap \mathcal{V} = \varnothing$.

Expressions and conditions are considered implementation-dependent. The set of all expressions is denoted by $\mathcal{E}$, while the set of all conditions is denoted by $\mathcal{B} \subseteq \mathcal{E}$. Variables that occur in a given expression can be collected using the function $vars : \mathcal{E} \to \wp(\mathcal{V})$. In the following, an expression is usually denoted through the symbol e, while φ and ψ are used to denote conditions.

A sequence of elements $\langle x_i \rangle_{i=1}^{n}$ will be often denoted by a symbol resembling the content with a bar on the top, e.g. $\bar{v} \triangleq \langle v_i \rangle_{i=1}^{n}$ for a sequence of variables, or $\bar{\varphi} \triangleq \langle \varphi_i \rangle_{i=1}^{n}$ for denoting a sequence of conditions.

As a result of the imposed syntactical rules, any process model expressed in NestFlow can be seen as a tree of control-flow elements in which internal nodes are non-terminal blocks and leafs are commands. Therefore, no one should be surprised that the most natural data structure for checking and interpreting a process model is an Abstract Syntax Tree (AST) [107]: an AST captures the essential data of the model elements and the relations among them, discarding all those aspects that are not relevant from the semantic perspective. Every AST node has a specific type and structure: node types are summarized in Tab. 5.1 together with a brief description. For convenience, symbol $\mathcal{T}\!\!\mathit{n}$ is used to denote the set containing all node types summarized in Tab. 5.1. The node structure is nothing more than a tuple $\langle x_i \rangle_{i=1}^{n}$ whose attributes are determined by the first one that corresponds to a note type, i.e. $x_1 \in \mathcal{T}$.

Definition 5.5 (Process Fragment). An AST node is a tuple containing at least a node type in the first position, followed by zero or more attributes. The type and position of each attribute depends on the initial node type. The set of all process fragments obtained assembling these AST nodes is denoted by $\mathcal{N}$ and recursively defined as follows:

$$
\begin{aligned}
&\forall x \in \mathcal{U} \,.\, x \in \mathcal{N} \mathrel{\hat{\Longleftrightarrow}} \exists a, b \in \mathcal{N} \,. \qquad (5.3)\\
&\quad (\exists T \in \mathcal{T} \,.\, \exists \bar{\alpha}, \bar{\beta} \in \mathcal{S}^* \,.\, \exists \bar{\xi} \in \mathcal{X}^* \,.\, \exists \iota : \mathcal{T} \to \mathcal{N} \,.\, x = \langle \textsf{PROC}, T, \iota, \bar{\alpha}, \bar{\beta}, \bar{\xi}, a \rangle) \vee\\
&\quad (\exists \bar{v} \in \mathcal{V}^* \,.\, \exists \bar{e} \in \mathcal{E}^* \,.\, x = \langle \textsf{LET}, \bar{v}, \bar{e}, a \rangle \wedge |\bar{v}| = |\bar{e}|) \vee\\
&\quad (\exists t \in \mathcal{I} \,.\, x = \langle \textsf{SEQ}, t, a, b \rangle) \vee\\
&\quad (\exists t \in \mathcal{I} \,.\, \exists \varphi \in \mathcal{B} \,.\, x = \langle \textsf{CHOICE}, t, \varphi, a, b \rangle) \vee\\
&\quad (\exists t \in \mathcal{I} \,.\, \exists \varphi \in \mathcal{B} \,.\, x = \langle \textsf{LOOP}, t, \varphi, a, b \rangle) \vee\\
&\quad (\exists t \in \mathcal{I} \,.\, \exists \xi \in \mathcal{X} \,.\, x = \langle \textsf{TRY}, t, \xi, a, b \rangle) \vee\\
&\quad (\exists t \in \mathcal{I} \,.\, \exists \psi \in \mathcal{B} \,.\, x = \langle \textsf{PAR}, t, u, w, \psi, a, b \rangle) \vee\\
&\quad (\exists t \in \mathcal{I} \,.\, \exists v \in \mathcal{V} \,.\, x = \langle \textsf{THRES}, t, v, a \rangle) \vee\\
&\quad (\exists t \in \mathcal{I} \,.\, x = \langle \textsf{CONCUR}, t, a \rangle) \vee\\
&\quad (x = \langle \textsf{SKIP} \rangle) \vee\\
&\quad (\exists t \in \mathcal{I} \,.\, T \in \mathcal{T} \,.\, \exists \bar{r}, \bar{o} \subseteq \mathcal{K} \times \mathcal{S} \times \mathcal{K} \times \mathcal{S} \,.\, x = \langle \textsf{RUN}, t, T, \bar{r}, \bar{o} \rangle) \vee\\
&\quad (\exists e \in \mathcal{E} \,.\, \exists T \in \mathcal{T} \,.\, \exists s \in \mathcal{I} \,.\, x = \langle \textsf{SPAWN}, e, T, s \rangle) \vee\\
&\quad (\exists \xi \in \mathcal{X} \,.\, \exists \bar{t} \in \mathcal{I}^* \,.\, x = \langle \textsf{THROW}, \xi, \bar{t} \rangle) \vee\\
&\quad (\exists t \in \mathcal{I} \,.\, \exists \bar{v}, \bar{\beta} \in \mathcal{S}^* \,.\, x = \langle \textsf{SEND}, t, \bar{v}, \bar{\beta} \rangle \wedge |\bar{v}| = \left|\bar{\beta}\right|) \vee\\
&\quad (\exists t \in \mathcal{I} \,.\, \exists \bar{v}, \bar{\alpha} \in \mathcal{S}^* \,.\, \exists \theta \in \mathcal{V} \,.\, x = \langle \textsf{RECEIVE}, t, \bar{v}, \bar{\alpha}, \theta \rangle \wedge |\bar{v}| = |\bar{\alpha}|) \vee\\
&\quad (\exists \alpha \in \mathcal{S} \,.\, x = \langle \textsf{EMPTY}, \alpha \rangle)
\end{aligned}
$$

where the name of the contained nodes $a, b \in \mathcal{N}$ are consistent with the non-terminal symbols $\langle A \rangle$ and $\langle B \rangle$ found in the grammar of Fig. 5.11 in Sec. 5.7, in particular a is the first encountered non-terminal symbol in the control-flow. □

In the **PROC** node definition symbols $\bar{\alpha}$, $\bar{\beta}$, and $\bar{\xi}$ denote the set of input streams, output streams and raised exceptions, respectively. Such sets are represented as sequences of elements to preserve their relative order. Conversely, the function

Table 5.1. AST node types and their meaning.

Type	Meaning	Type	Meaning	Type	Meaning
PROC	process	**SEQ**	sequence block	**SKIP**	skip command
		CHOICE	choice block	**RUN**	run command
LET	variables	**LOOP**	loop block	**SPAWN**	spawn command
	declaration	**TRY**	try-catch block	**THROW**	throw command
		PAR	parallel block	**SEND**	send command
		THRES	threshold block	**RECEIVE**	receive command
		CONCUR	concurrent block	**EMPTY**	empty command

$\iota : \mathcal{T} \to \mathcal{N}$ identifies the declared import associated to a given type. Similarly, in the definition of the **LET** node, sequence $\bar{v}$ represents the set of process variables, while sequence $\bar{e}$ is the set of initialization expressions, one for each variable. For sake of simplicity, it is assumed that an initialization expression in $e \in \bar{e}$ is a constant or null value; hence, it cannot contain other variables. In the remaining node definitions, symbol t denotes the unique identifier associated to a non-terminal block or a command. Notice that an identifier is associated also to the **SEQ** node even if it is not graphically represented in the grammar. The join condition ψ of a **PAR** node is optional and can be represented by the empty sequence ε. The **RUN** node definition contains the tuples $\bar{o} \subseteq \mathcal{K}\times\mathcal{S}\times\mathcal{K}\times\mathcal{S}$ and $\bar{r} \subseteq \mathcal{K}\times\mathcal{S}\times\mathcal{K}\times\mathcal{S}$ which identify the input and output stream mappings, respectively. Each input mapping contains the process instance identifier, the identifer of its involved input stream, the identifier of another process instance, and the identifier of an output stream of this last one. Each output mapping is defined in a similar way. In the definition of the **SPAWN** node, the expression e is a simple expression used to denote a storing location for the identifier of the dynamically generated instance. The symbol T denotes the type of the spawned process, while the last identifier s is used to specify the target `concur` scope when necessary. The definition of the **THROW** node contains a sequence of identifiers $\bar{t}$ that can be optionally used for specifying a set of parallel branches on which the exception is thrown. In the **SEND** and **RECEIVE** nodes sequence $\bar{v}$ denotes a set of variables, while $\bar{\beta}$ and $\bar{\alpha}$ are streams. Clearly, the number of specified variables and streams has to be the same. Finally, the value of variable θ in the **RECEIVE** node denotes a timeout after which the command is interrupted. The **EMPTY** node has a single parameter representing the name of an existing stream whose content has to be discarded.

Some useful functions can be defined on process fragments. In particular, the type of the root node is captured by the function $type : \mathcal{N} \to \mathcal{T}\!n$, which returns the first element of the tuple, namely for each $x = \langle a_i \rangle_{i=1}^{n} \in \mathcal{N}$, $type(x) = a_1$. The function $children : \mathcal{N} \to \mathcal{N}^*$ defined in Eq. 5.4 returns the list of child nodes if the argument is non-terminal block, the empty sequence otherwise.

$$children : \mathcal{N} \to \mathcal{N}^* \ \textit{is} \tag{5.4}$$

$$\forall x \in \mathcal{N} \,.\, children(x) = \begin{cases} \langle y, z \rangle & \text{if } \exists y, z \in \mathcal{N} \,. \\ & \quad x = \langle \textsf{SEQ}, \_, y, z \rangle \vee \\ & \quad x = \langle \textsf{CHOICE}, \_, \_, y, z \rangle \vee \\ & \quad x = \langle \textsf{LOOP}, \_, \_, y, z \rangle \vee \\ & \quad x = \langle \textsf{TRY}, \_, \_, y, z \rangle \vee \\ & \quad x = \langle \textsf{PAR}, \_, \_, y, z \rangle \\ \langle y \rangle & \text{if } \exists y \in \mathcal{N} \,. \\ & \quad x = \langle \textsf{PROC}, \_, \_, \_, \_, y \rangle \vee \\ & \quad x = \langle \textsf{LET}, \_, \_, y \rangle \vee \\ & \quad x = \langle \textsf{THRES}, \_, \_, y \rangle \vee \\ & \quad x = \langle \textsf{CONCUR}, \_, y, \_ \rangle \\ \varepsilon & \text{otherwise} \end{cases}$$

Recalling from the background, the elision symbol "_", used to match the node structure, means that there exists an element $u \in \mathcal{U}$ in the given position that does not need to be mentioned, e.g. $x = \langle \textsf{SEQ}, \_, y, z \rangle$ means $\exists u \in \mathcal{U} \,.\, x = \langle \textsf{SEQ}, u, y, z \rangle$ where u does not appear in the enclosing expression.

For those elements with only one child, the partial function $body : \mathcal{N} \rightharpoonup \mathcal{N}$ can be used to obtain it, i.e. $\forall x, y \in \mathcal{N} \,.\, body(x) = y \mathrel{\stackrel{\triangle}{\Longleftrightarrow}} children(x) = \langle y \rangle$. Function $nodes : \mathcal{N} \to \wp(\mathcal{N})$ returns the set of all nodes that belong to the specified root. It can be recursively defined as follows:

$$nodes : \mathcal{N} \to \wp(\mathcal{N}) \ \textit{is} \tag{5.5}$$
$$\forall x, y \in \mathcal{N} \,.\, y \in nodes(x) \mathrel{\stackrel{\triangle}{\Longleftrightarrow}} y = x \vee \exists z \in nodes(x) \,.\, y \in children(z)$$

Sometimes the children of a non-terminal block are referred through the partial functions $first : \mathcal{N} \rightharpoonup \mathcal{N}$ and $last : \mathcal{N} \rightharpoonup \mathcal{N}$. More specifically, for each $x \in \mathcal{N}$ such that $children(x) = \langle y_i \rangle_{i=1}^{k}$, the function $first(x) = y_1$ and $last(x) = y_k$ if $k \geq 1$, undefined otherwise. Whenever the node $x \in \mathcal{N}$ has a single body, $first(x) = last(x)$.

The function $id : \mathcal{N} \rightharpoonup \mathcal{I}$ defined in Eq. 5.6 returns the identifier assigned to a non-terminal block or command. When such identifier exists, it is placed just after the node type, hence the access function can be given as in Eq. 5.6, where $L = \{\textsf{SEQ}, \textsf{CHOICE}, \textsf{LOOP}, \textsf{TRY}, \textsf{PAR}, \textsf{THRES}, \textsf{CONCUR}, \textsf{RUN}, \textsf{SEND}, \textsf{RECEIVE}\}$ is simply the set containing all node types having an identifier.

$$id : \mathcal{N} \rightharpoonup \mathcal{I} \ \textit{is} \tag{5.6}$$
$$\forall x \in \mathcal{N} \,.\, id(x) = \begin{cases} a_2 & \text{if } \exists \langle a_i \rangle_{i=1}^{n} \in \mathcal{U} \,.\, x = \langle a_i \rangle_{i=1}^{n} \wedge n > 1 \ \wedge \ a_1 \in L \\ \uparrow & \text{otherwise} \end{cases}$$

The static components of a process are all those process instances invoked with a run, receive, or send command. The function $components : \mathcal{N} \to \wp(\mathcal{K})$, also denoted as $\kappa : \mathcal{N} \to \wp(\mathcal{K})$, returns the set of identifiers representing the static components of a process fragment.

$$
\begin{aligned}
&\mathit{components} : \mathcal{N} \to \wp(\mathcal{K}) \ \mathit{is} \\
&\quad \forall x \in \mathcal{N} \,.\, \forall t \in \mathcal{I} \,.\, \forall T \in \mathcal{T} \,.\, \langle t, T \rangle \in \kappa(x) \mathrel{\stackrel{\triangle}{\Longleftrightarrow}} \\
&\qquad \exists y \in \mathit{nodes}(x) \,.\, y = \langle \textsf{RUN}, t, T, \bar{r}, \bar{o} \rangle \vee \\
&\qquad\qquad (y = \langle \textsf{SEND}, t, \bar{v}, \bar{\beta} \rangle \wedge T = \textsf{S}) \vee \\
&\qquad\qquad (y = \langle \textsf{RECEIVE}, t, \bar{v}, \bar{\alpha}, \theta \rangle \wedge T = \textsf{R})
\end{aligned}
\tag{5.7}
$$

Using this definition the set of process components $\pi : \mathcal{N} \to \wp(\mathcal{K})$ can be derived as follows: $\forall x \in \mathcal{N} \,.\, \pi(x) = \{y \in \kappa(x) \mid \mathit{type}(y) \notin \{\textsf{S}, \textsf{R}\}\}$.

Definition 5.6 (Referred Variables). The set of variables used inside a process fragment can be computed by the recursive function $\mathit{vars} : \mathcal{N} \to \wp(\mathcal{V})$ defined as:

$$
\mathit{vars} : \mathcal{N} \to \wp(\mathcal{V}) \ \mathit{is} \ \forall x \in \mathcal{N} \,. \tag{5.8}
$$

$$
\mathit{vars}(x) = \begin{cases}
\mathit{vars}(\mathit{body}(x)) & \text{if } \mathit{type}(x) \in \{\textsf{PROC}, \textsf{LET}, \textsf{CONCUR}\} \\
\mathit{vars}(y) \cup \mathit{vars}(z) & \text{if } \mathit{type}(x) \in \{\textsf{SEQ}, \textsf{TRY}\} \wedge \\
 & \quad \wedge \exists y, z \in \mathcal{N} \,.\, \langle y, z \rangle = \mathit{children}(x) \\
\mathit{vars}(\varphi) \cup & \text{if } \exists y, z \in \mathcal{N} \,.\, \exists \varphi \in \mathcal{B} \,. \\
\mathit{vars}(y) \cup \mathit{vars}(z) & \quad (\ x = \langle \textsf{CHOICE}, \_, \varphi, y, z \rangle \vee \\
 & \qquad x = \langle \textsf{LOOP}, \_, \varphi, y, z \rangle \vee \\
 & \qquad x = \langle \textsf{PAR}, \_, \varphi, y, z \rangle \ ) \\
\{v\} \cup \mathit{vars}(y) & \text{if } \exists y \in \mathcal{N} \,.\, \exists v \in \mathcal{V} \,.\, n = \langle \textsf{THRES}, \_, v, y \rangle \\
\mathit{vars}(e) & \text{if } \exists e \in \mathcal{E} \,.\, x = \langle \textsf{SPAWN}, e, \_, \_ \rangle \\
\mathit{set}(\bar{v}) & \text{if } \exists \bar{v} \in \mathcal{V}^* \,.\ x = \langle \textsf{SEND}, \_, \bar{v}, \_ \rangle \\
\mathit{set}(\bar{v}) \cup \{\theta\} & \text{if } \exists \bar{v} \in \mathcal{V}^* \,.\, \exists v \in \mathcal{V} \,.\, x = \langle \textsf{RECEIVE}, \_, \bar{v}, \_, \theta \rangle \\
\varnothing & \text{otherwise}
\end{cases}
$$

□

Referenced variables should not be confused with the declared ones. The function $\mathit{declared\text{-}vars} : \mathcal{N} \to \wp(\mathcal{V})$ returns the variables declared in a process fragment:

$$
\begin{aligned}
&\mathit{declared\text{-}vars} : \mathcal{N} \to \wp(\mathcal{V}) \ \mathit{is} \\
&\quad \forall x \in \mathcal{N} \,.\, \forall V \subseteq \mathcal{V} \,.\, \mathit{declared\text{-}vars}(x) = V \mathrel{\stackrel{\triangle}{\Longleftrightarrow}} \\
&\qquad \exists \bar{v} \in \mathcal{V}^* \,.\, \exists \bar{e} \in \mathcal{E}^* \,.\, \exists y \in \mathcal{N} \,.\, x = \langle \textsf{LET}, \bar{v}, \bar{e}, y \rangle \wedge V = \mathit{set}(\bar{v})
\end{aligned}
\tag{5.9}
$$

A process model can be defined as a tree having a root of type **PROC**. However, for sake of simplicity, it is also required that all variables are declared at the beginning of a process. This is not a big limitation because any model can be transformed into an equivalent one with all variables declared at the beginning by renaming those variables contained in a different scope having the same name.

Definition 5.7 (Process Model). The set of process models $\mathit{Nest} \subseteq \mathcal{N}$ is given by all process fragments having a unique **PROC** root followed by a unique **LET** node.

$$
\begin{aligned}
&\forall x \in \mathcal{N} \,.\, x \in \mathit{Nest} \mathrel{\stackrel{\triangle}{\Longleftrightarrow}} \\
&\quad \mathit{type}(x) = \textsf{PROC} \wedge \\
&\quad \exists y \in \mathcal{N} \,.\, y = \mathit{body}(x) \wedge \mathit{type}(y) = \textsf{LET} \wedge \\
&\quad \forall z \in \mathit{nodes}(\mathit{body}(y)) \,.\, \mathit{type}(z) \notin \{\textsf{PROC}, \textsf{LET}\}
\end{aligned}
\tag{5.10}
$$

□

The interface of a process $x \in \mathcal{N}est$ can be characterized by three similar functions $pin : \mathcal{N}est \to \mathcal{S}^*$, $pout : \mathcal{N}est \to \mathcal{S}^*$, and $pex : \mathcal{N}est \to \mathcal{X}^*$ returning the declared list of input streams, output streams, and exceptions, respectively. For example, $pin : \mathcal{N}est \to \mathcal{S}^*$ can be defined as $\forall x \in \mathcal{N}est \,.\, \forall \bar{\alpha} \in \mathcal{S}^* \,.\, pin(x) = \bar{\alpha} \mathrel{\triangleq\!\!\Longleftrightarrow} x = \langle \textsf{PROC}, \_, \_, \bar{\alpha}, \_, \_, \_ \rangle$.

The model of a process instance depends on the set of imported definitions. The function $model : \mathcal{N}est \times \mathcal{T} \to \mathcal{N}est$ returns the model of the specified type imported in the given one. It is formally defined as follows:

$$\begin{aligned} &model : \mathcal{N}est \times \mathcal{T} \to \mathcal{N}est \ \textit{is} \\ &\quad \forall x, y \in \mathcal{N}est \,.\, \forall T \in \mathcal{T} \,.\, model(x, T) = y \mathrel{\triangleq\!\!\Longleftrightarrow} \\ &\quad\quad x = \langle \textsf{PROC}, \_, \iota, \_, \_, \_, \_ \rangle \ \wedge \ \iota(T) = y. \end{aligned} \tag{5.11}$$

Example 5.8. The process model P depicted in Fig. 5.41 is a structured NestFlow representation of the model in Fig. 5.2. The two cycles synchronize each other at every iteration through message passing, and the two loop conditions are evaluated on distinct variable sets, avoiding race conditions.

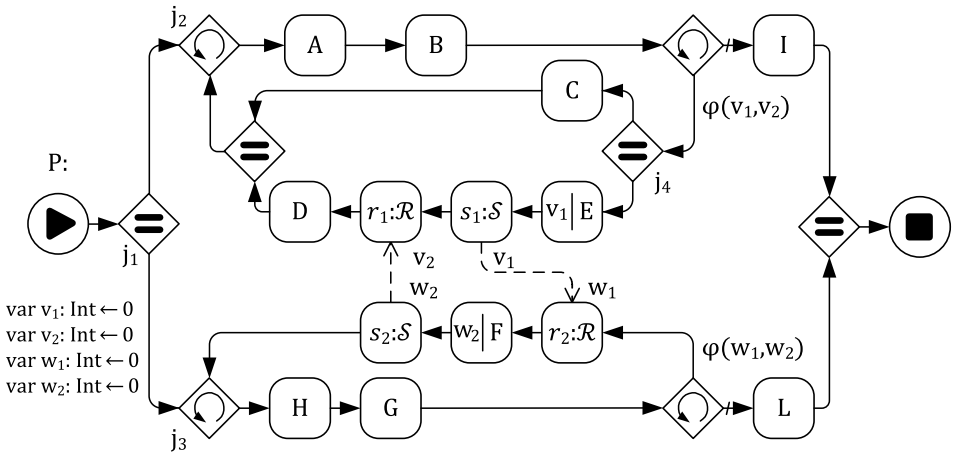


Fig. 5.41. A structured NestFlow process model.

Fig. 5.42 illustrates the AST of the given model: for each node, the left branch always contains the unfolding of the a children, and symmetrically the right branch contains the unfolding of the b children.

In the **PROC** node, the input streams, output streams and raised exceptions are denoted by the empty string ε because no one of them is declared by the process, while the ι function is represented by an empty set. The variables written by E and F are captured by a subsequent **RECEIVE** node not explicitly represented in the model of Fig. 5.41. Finally, the ω symbol in each **RECEIVE** node denotes that no timeout is defined for them. □

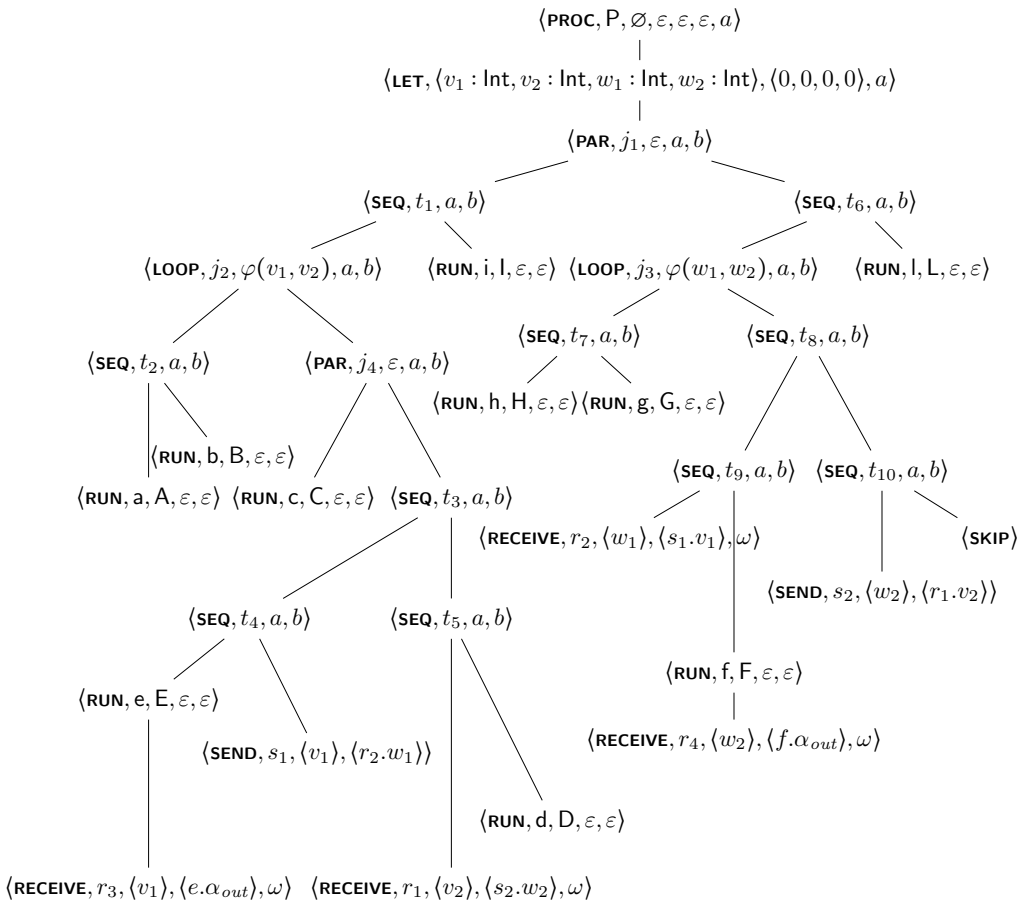


Fig. 5.42. The AST of the model in Fig. 5.41.

5.9.2 Well-Formedness Properties

Well-formedness properties are additional properties stated for discerning *valid* process models from those that are clearly erroneous from a syntactical point of view. These properties concern the behavior of a process model, but they can be statically checked without executing it. Therefore, they guarantee the absence of certain run-time errors that may be hard to prove otherwise. For example, a well-formedness property can state that a model is valid only if all used variables are declared in advance and visible in the scope in which they are referenced: $\mathit{vars}(x) \subseteq \mathit{declared\text{-}vars}(x)$ for all valid $x \in \mathcal{N}\mathit{est}$.

A type system is the primary example of well-formedness: if types in a model are coherent, then many faulty states cannot be reached when it is executed. This section introduces some important well-formedness properties of NestFlow that can be considered unusual with respect to other languages. In particular, two main properties have to be enforced in a NestFlow model: every process instance can be invoked in only one place in the model and no variable can be shared among parallel branches. These properties are formalized by the following definitions that also show how they can be computed.

Definition 5.9 (One Place per Process Instance). A process $x \in \mathcal{N}est$ is valid only if every task instance $t:T$ in x has a unique identifier $t \in \mathcal{I}$ and is executed in only one place. Such property can be captured by the $uniqueness : \mathcal{N}est \to \mathbb{B}$ predicate defined as follows:

$$uniqueness(x) \stackrel{\Delta}{\iff} \forall c \in occurrences(x) \,.\, \mu(c) = 1 \tag{5.12}$$

where the function $occurrences : \mathcal{N} \to \mathcal{M}$, also denoted as $\delta : \mathcal{N} \to \mathcal{M}$, computes the set of static occurrences that appear in a process $x \in \mathcal{N}est$. The set of static occurrences is a multiset containing the pairs $\langle t, i \rangle$, made of an instance identifier $t \in \mathcal{I}$ and a number $i \in \mathbb{N}$ counting how many times t appears in the model.

$$\begin{aligned} &occurrences : \mathcal{N} \to \mathcal{M} \ \textit{is} \\ &\quad \forall x \in \mathcal{N} \,. \\ &\qquad \delta(x) = \begin{cases} \delta(body(x)) & \text{if } type(x) \in \{\textsf{PROC}, \textsf{LET}, \textsf{THRES}, \textsf{CONCUR}\} \\ \delta(y) \cup \delta(z) & \text{if } type(x) \in \{\textsf{SEQ}, \textsf{CHOICE}, \textsf{LOOP}, \textsf{TRY}, \textsf{PAR}\} \ \wedge \\ & \qquad \wedge \ \exists y, z \in \mathcal{N} \,.\, \langle y, z \rangle = children(x) \\ multiset(id(x)) & \text{if } type(x) \in \{\textsf{RUN}, \textsf{SEND}, \textsf{RECEIVE}\} \\ \varnothing & \text{otherwise} \end{cases} \end{aligned} \tag{5.13}$$

□

When $uniqueness(x)$ holds for a process model $x \in \mathcal{N}est$, there is a unique correspondence between occurrences and process instances. This property guarantees that there are no concurrent executions of the same component which may leave the process in an inconsistent state. It can be relaxed allowing sequential invocations of the same process instance, but taken as is, it also ensures a unique graphical representation of senders and receivers, i.e. it is always clear where are source and target of a link.

Definition 5.10 (No Shared Variables). A process model $x \in \mathcal{N}est$ is valid only if its parallel branches do not share variables. Such property is captured by the predicate $isolation : \mathcal{N} \to \mathbb{B}$, also denoted by $\eta : \mathcal{N} \to \mathbb{B}$, that is true when no variable is used in two different places inside distinct parallel branches.

$$isolation : \mathcal{N} \to \mathbb{B} \ is \tag{5.14}$$
$$\forall x \in \mathcal{N} .$$
$$\eta(x) = \begin{cases} \eta(body(x)) & \text{if } type(x) \in \{\textsf{PROC}, \textsf{LET}, \textsf{THRES}, \textsf{CONCUR}\} \\ \eta(y) \wedge \eta(z) & \text{if } type(x) \in \{\textsf{SEQ}, \textsf{CHOICE}, \textsf{LOOP}, \textsf{TRY}\} \wedge \\ & \quad \wedge \exists y, z \in \mathcal{N} . \langle y, z \rangle = children(x) \\ (vars(y) \cap vars(z) = \varnothing) \wedge & \text{if } type(x) = \textsf{PAR} \wedge \\ \wedge \eta(y) \wedge \eta(z) & \quad \wedge \exists y, z \in \mathcal{N} . \langle y, z \rangle = children(x) \\ \textsf{T} & \text{otherwise} \end{cases}$$

□

This property prunes away race conditions or non-deterministic behaviors caused by the exact timing of events that are not completely under the designer's control. This rule can be relaxed for those variables that are not concurrently updated, i.e. they do not appear as left-values in assignments performed by parallel branches. In the bare language this can be realized by checking where objects are stored in a receive command, since this is the only place in which variables are updated.

5.9.3 Data-flow Subsumption

AMP is the only abstraction offered by the bare language to exchange data among processes. As previously discussed, this is reasonable in a minimal language because AMP can simulate other mechanisms, like parameter passing. This section discusses how the links in a model can be substantially reduced to improve readability. The focus is only on static relations among tasks, namely relations that can be explicitly represented through the grammar constructs. Dynamic relations are also important, but they are not treated here because in the bare language they lack of a graphical representation.

Definition 5.11 (Control-flow Relation). The function *static control-flow relation* $cf : \mathcal{N} \to \wp(\mathcal{K} \times \mathcal{K})$ returns for each process fragment $x \in \mathcal{N}$ the pairs of its internal task instances $\langle i, j \rangle \in \kappa(x) \times \kappa(x)$ between which a control-flow relation exists. It is defined as follows using two auxiliary functions $\rho(x) = \{\langle a, a \rangle \mid a \in \kappa(x)\}$ and $\gamma(y, z) = \{\langle a, b \rangle \mid a \in \kappa(y) \wedge b \in \kappa(z)\}$ as follows:

$$cf : \mathcal{N} \to \wp(\mathcal{K} \times \mathcal{K}) \ is \tag{5.15}$$
$$\forall x \in \mathcal{N} .$$
$$cf(x) = \begin{cases} cf(body(x)) & \text{if } type(x) \in \{\textsf{PROC}, \textsf{LET}, \textsf{THRES}, \textsf{CONCUR}\} \\ cf(y) \cup cf(z) & \text{if } type(x) \in \{\textsf{CHOICE}, \textsf{TRY}, \textsf{PAR}\} \wedge \\ & \quad \wedge \exists y, z \in \mathcal{N} . \langle y, z \rangle = children(x) \\ cf(y) \cup cf(z) \cup \tau(y, z) & \text{if } type(x) = \textsf{SEQ} \wedge \\ & \quad \wedge \exists y, z \in \mathcal{N} . \langle y, z \rangle = children(x) \\ cf(y) \cup cf(z) \cup \rho(y) \cup \rho(z) \cup & \text{if } type(x) = \textsf{LOOP} \wedge \\ \quad \cup \gamma(y, z) \cup \gamma(z, y) & \quad \wedge \exists y, z \in \mathcal{N} . \langle y, z \rangle = children(x) \\ \varnothing & \text{otherwise} \end{cases}$$

□

When $x \in \mathcal{N}est$ is clear from the context, the notation $y \to z$ can be used in place of $\langle y, z\rangle \in cf(x)$, which stands for y executes before z. Let us notice that $t \to t$ is not a contradiction, it simply means that an execution of an instance t can be followed by another execution of the same instance.

Definition 5.12 (Internal Streams and Links). The set of *internal static input streams* available in a process model $x \in \mathcal{N}est$ is given by the union of its interface input streams with the output streams of its static components. The set of *internal static output streams* is defined in similar way. They can be respectively denoted using the following functions $lin : \mathcal{N}est \to \wp(\mathcal{I}\times\mathcal{S})$ and $lout : \mathcal{N}est \to \wp(\mathcal{I}\times\mathcal{S})$:

$$lin : \mathcal{N}est \to \wp(\mathcal{I}\times\mathcal{S}) \ \textit{is} \tag{5.16}$$
$$\forall x \in \mathcal{N}est\,.$$
$$lin(x) = \{\langle \varsigma, \alpha\rangle \mid \alpha \in pin(x)\} \bigcup_{\langle t,T\rangle\in\kappa(x)} \{\langle t, \beta\rangle \mid \beta \in pout(model(x,T))\}$$

$$lout : \mathcal{N}est \to \wp(\mathcal{I}\times\mathcal{S}) \ \textit{is} \tag{5.17}$$
$$\forall x \in \mathcal{N}est\,.$$
$$lout(x) = \{\langle \varsigma, \alpha\rangle \mid \alpha \in pout(x)\} \bigcup_{\langle t,T\rangle\in\kappa(x)} \{\langle t, \beta\rangle \mid \beta \in pin(model(x,T))\}$$

□

The dot notation $t.\alpha$ is used to denote a pair $\langle t, \alpha\rangle$ that belongs to the internal streams $lin(x)$ or $lout(x)$ of a process $x \in \mathcal{N}est$, where the special symbol ς stands for the optional keyword this.

A *static data-flow relation* $df(x) \subseteq lin(x)\times lout(x)$ on a process model $x \in \mathcal{N}est$ can be used for representing links between internal streams: an implementation may use this static relation and its dynamic counterpart for routing objects among streams. Two streams $\alpha : A$ and $\beta : B$ can be connected by a link $\langle \alpha, \beta\rangle \in df(x)$ only if A and B are two compatible types. For sake of simplicity, in the following it is assumed that the two streams have the same type $T = A = B$. Whenever the model $x \in \mathcal{N}est$ is clear from the context, the notation $\alpha \overset{T}{\dashrightarrow} \beta$ can be used in place of $\langle \alpha, \beta\rangle \in df(x)$, where T is the common type of α and β, or simply $\alpha \dashrightarrow \beta$ if the type is not relevant for the discussion.

The use of AMP for representing all data-flow dependencies in a model can soon become verbose. However, NestFlow allows one to hide links and their related commands anywhere they can be subsumed by the control-flow. When stream identifiers are not relevant, links with the same source and target can be grouped into a unique dashed arrow, called *collapsed link*. During the design of a task, loopback external links and outgoing external links can be safely ignored, but in general incoming external links are important, because they denote an external resource needed to continue the execution. In order to determine the relevance of the internal links of a model $x \in \mathcal{N}est$, the static control-flow relation $cf(x)$ can be compared with the coarse-grained data-flow relation $gf(x)$ defined below.

Definition 5.13 (Coarse-grained data-flow relation). The *coarse-grained data-flow relation* $gf : \mathcal{N}est \to \mathcal{K}\times\mathcal{K}$ is a function that returns the pairs of components $\langle y, z\rangle \subseteq \kappa(x)\times\kappa(x)$ among which a collapsed link exists.

$$
\begin{aligned}
&gf : \mathcal{N}est \to \mathcal{K}\times\mathcal{K} \ \mathit{is} \\
&\quad \forall x \in \mathcal{N}est \,. \\
&\qquad gf(x) = \{\langle y, z\rangle \mid y, z \in \kappa(x) \ \wedge \ \exists \beta \in \mathit{pout}(y) \,. \\
&\qquad\quad \exists \alpha \in \mathit{pin}(z) \,. \langle y.\beta, z.\alpha\rangle \in \mathit{df}(x)\}
\end{aligned}
\tag{5.18}
$$

□

Given a model $x \in \mathcal{N}est$, any pair $\langle y, z\rangle \in gf(x)$ can also be denoted as $y \dashrightarrow z$ using the same dashed arrow adopted for streams, but now applied to components.

Given a model $x \in \mathcal{N}est$, the relation $gf(x)$ contains only collapsed links and excludes all external links because $\varsigma \notin \kappa(x)$. The relation $y \dashrightarrow z$ means that y may send one or more objects to z through some underlying streams. The relation $gf(x)$ is a good approximation of $df(x)$, because collapsed links share the same source and destination of the subsumed links.

For determining the relevance of a data-flow relation $\langle y, z\rangle \in gf(x)$, it is compared with the existing control-flow relations between y and z. In particular, four cases can be distinguished:

1. $y \to z \wedge \neg(z \to y)$: the relation $y \dashrightarrow z$ matches the control-flow path and can be subsumed without loosing critical information.
2. $y \to z \wedge z \to y$: the situation is the same as the previous point but inside a loop, hence the relation $y \dashrightarrow z$ can be safely subsumed.
3. $\neg(y \to z) \wedge z \to y$: there is a potential error because z may have no chance to use the object received from y. The model may be correct if z is a stateful component that does not require the object sent by y during one of its executions.
4. $\neg(y \to z) \wedge \neg(z \to y)$: relation $y \dashrightarrow z$ is essential for describing task dependencies among parallel branches and has to be explicitly specified in the model. Actually, this last case can also occur with links between branches of other constructs like choice or try blocks, but these branches are executed in mutual exclusion and they can communicate through shared variables; in such case explicit links can be omitted.

Following these considerations, data-flow relations explicitly represented in a model can be substantially reduced to those that are relevant for control-flow design, while the other ones can be hidden for increasing readability. The remaining links mostly describe interactions among concurrent entities. Moreover, data-flow relations that are in contrast with the control-flow structure can be used for highlighting possible errors but a further analysis is required in presence of composite tasks.

This gives an idea on how links can be minimized for not cluttering the diagram. An actual implementation can introduce further abstractions for exchange data among components, like parameter passing. Nevertheless, AMP remains the right abstraction for describing the interaction of concurrent entities.

5.9.4 NestFlow Interpreter

The procedure given from Lis. 5.1 to Lis. 5.8 contains the main steps of an interpreter for the NestFlow bare language. The overall interpreter architecture is not complex, considering the features offered by the language: it is made of a scheduler that selects which thread to activate, and a decoder that matches and executes each single instruction producing the desired effects.

The interpreter uses two simple data structures: stack and queue, which are defined as usual. The set of all stacks is denoted as $\mathcal{S}tack$, while the set of all queues is denoted as $\mathcal{Q}ueue$. The operations on those data structures have the usual name, e.g. $top : \mathcal{S}tack \to \mathcal{U}$ returns the first element of a stack, $pop : \mathcal{S}tack \to \mathcal{S}tack$ removes the first stack element, and $push : \mathcal{S}tack \times \mathcal{U} \to \mathcal{S}tack$ adds an element on the top of the current stack. Similarly, $head : \mathcal{Q}ueue \to \mathcal{U}$ returns the head of a queue, while $tail : \mathcal{Q}ueue \to \mathcal{Q}ueue$ returns its tail, and $enqueue : \mathcal{Q}ueue \times \mathcal{U} \to \mathcal{Q}ueue$ enqueues an object to the current queue.

Each thread of control has its own stack handled as a continuation containing the instructions to be executed. Variables and streams are managed through a data structure, called *store*, that relates such entities with their current state. A snapshot of the store can be represented through function $\sigma : \mathcal{V} \cup \mathcal{S} \rightharpoonup \mathcal{U}$ that associates to each variable or stream its pointed object. Certain instructions update the store content and its representation has to change accordingly. The set of all store functions $\sigma : \mathcal{V} \cup \mathcal{S} \rightharpoonup \mathcal{U}$ is denoted by Σ. How the store is managed and what it contains are considered implementation details and so not treated here.

In the following it is assumed the existence of the function $eval : \Sigma \times \mathcal{E} \to \mathcal{U}$ which evaluates a given expression using the current store content. Furthermore, the function $update\text{-}store : \Sigma \times \mathcal{V} \times \mathcal{E} \to \Sigma$ is used to emphasize the store update and is defined as follows, where $\sigma(x)\downarrow$ stands for $\exists y \,.\, \sigma(x) = y$:

$$
\begin{aligned}
&update\text{-}store : \Sigma \times \mathcal{V} \times \mathcal{E} \to \Sigma \ \ is \\
&\quad \forall \sigma, \delta \in \Sigma \,.\, \forall v \in \mathcal{V} \,.\, \forall e \in \mathcal{E} \,.\, update\text{-}store(\sigma, v, e) = \delta \stackrel{\triangle}{\Longleftrightarrow} \\
&\qquad \forall x \in \mathcal{U} \,.\, \delta(x) = \begin{cases} \sigma(x) & \text{if } x \neq v \wedge \sigma(x)\downarrow \\ eval(\sigma, e) & \text{if } x = v \\ \uparrow & \text{otherwise} \end{cases}
\end{aligned}
\tag{5.19}
$$

A *process context* is a tuple $\langle id, p, \sigma, \bar{s}, \langle g, l \rangle \rangle$ such that $id \in \mathcal{I}$ is a unique identifier for the context, $p \in \mathcal{N}est$ is a process model, $\sigma \in \Sigma$ is the current state of the store, $\bar{s} \in \mathcal{Q}ueue$ is a queue of labelled stacks, i.e. pairs $\langle i, s \rangle \in \mathcal{I} \times \mathcal{S}tack$ where i is a stack identifier and s is a stack, and the pair $\langle g, l \rangle \in \mathcal{I} \times \mathbb{N}$ is used to manage the effects of a threshold block. More specifically, a complete management of the threshold block requires to consider the concept of work-list, such that any time the end-user selects one of the tasks subject to the threshold, the other ones can be suspended until its completion. The following interpreter contains only the instructions for creating the group of contexts associated to the same threshold. The set of all pairs $\langle g, l \rangle \in \mathcal{I} \times \mathbb{N}$ used for defining threshold groups is denoted by $\mathcal{H}$, while the set of all process contexts is denoted by $\mathcal{C}xt \triangleq \mathcal{I} \times \mathcal{N}est \times \Sigma \times \mathcal{Q}ueue \times \mathcal{H}$. The context of a process is created by the function $context : \mathcal{I} \times \mathcal{N}est \times \mathcal{H} \to \mathcal{C}xt$ such that $\forall id \in \mathcal{I} \,.\, \forall x \in \mathcal{N}est \,.\, \forall \langle g, l \rangle \in \mathcal{H} \,.\, context(id, x, g, l) = \langle id, x, \varnothing, \varepsilon, \langle g, l \rangle \rangle$.

Listing 5.1 NestFlow interpreter (1/8): initialization and main process nodes.

input: A process model $p \in \mathcal{N}est$ to be interpreted.

```
NestFlow-Interpreter(p)
 1  q ← enqueue(ε, context(ε, p, ε, ω));
 2  while ¬ halt(find(q, ε)) do
 3      q ← update-streams(q);
 4      ⟨cn, p, σ, s̄, ⟨g, l⟩⟩ ← head(q);
 5      q ← tail(q);
 6      if limited(g, l) then
 7          q ← enqueue(q, ⟨cn, p, σ, s̄, ⟨g, l⟩⟩);
 8          continue
 9      end if
10      z̄ ← ε;
11      while ¬ is-empty(s̄) do
12          ⟨d, h⟩ ← head(s̄);
13          s̄ ← tail(s̄);
14          inst ← top(h);
15          h ← pop(h);
16          match inst with
17              item ⟨PROC, T, ι, ᾱ, β̄, ξ̄, a⟩ do
18                  for each γ ∈ ᾱ ∪ β̄ do
19                      σ ← update-store(σ, γ, ε);
20                  end for
21                  h ← push(h, ⟨HALT, ε⟩);
22                  h ← push(h, a);
23                  z̄ ← enqueue(z̄, ⟨d, h⟩);
24              item ⟨HALT, t⟩ do
25                  h ← push(h, inst);
26                  z̄ ← enqueue(z̄, ⟨d, h⟩);
27              item ⟨LET, v̄, ē, a⟩ do
28                  let v̄ = {v_i}_{i=1}^{n}, ē = {e_j}_{j=1}^{m};
29                  for i ← 1 to n do
30                      σ ← update-store(σ, v_i, eval(σ, e_i));
31                  end for
32                  h ← push(h, a);
33                  z̄ ← enqueue(z̄, ⟨d, h⟩);
          ...continue...
```

The initial part of the interpreter receives in input a NestFlow process model p to be interpreted, then it creates a new process context for p using an empty store and an empty queue of stacks. This context is added to the queue q of process contexts. The procedure iterates through the created process contexts offering the chance to execute an instruction to each contained thread. The procedure continues by considering a process model $p \in \mathcal{N}est$, a store $\sigma \in \Sigma$, a list of threads of control $\bar{s} \in \mathcal{S}tack^*$, and a queue $q \in \mathcal{C}xt^*$ with the remaining process contexts. The tuple $\langle cn, p, \sigma, \bar{s}, \langle g, l \rangle \rangle$ represents the current process context. The procedure iterates

through the list of stacks $\bar{s}$ in order to execute their content an instruction at a time. The main steps of the procedure can be summarized as follows:

1 Initialization of the main context related to the process p. The identifier of the main context is ε, while the threshold group is represented by the pair $\langle \varepsilon, \omega \rangle$, where ω stands for an unlimited number of concurrent executions.

2-228 Main loop of the procedure: the procedure iterates for each context contained in the queue q, until the main context has something to do. The function $find : \mathcal{Q}ueue \times \mathcal{I} \to \mathcal{C}xt$ returns the context with the specified identifier in the given queue, while the function $halt : \mathcal{C}xt \to \mathbb{B}$ returns true if the context has only one stack and this one contains only an **HALT** instruction.

3-10 At each iteration of the main loop, the connected streams in the various contexts are synchronized each other through the function $update\text{-}streams : \mathcal{C}xt^* \to \mathcal{C}xt^*$, then the first context in q is retrieved and removed from the queue. If this context cannot be processed due to a thread threshold limit l associated to its group g, it is enqueued again and the interpreter considers the next available context. The function $limited : \mathcal{I} \times \mathbb{N} \to \mathbb{B}$ returns true if the specified group of contexts has reached the given thread threshold, false otherwise. If the context is not limited, it can be processed and a new queue $\bar{z}$ is initialized for storing its updated stacks.

11-226 The inner loop considers all the stacks associated to the given context and execute the first instruction contained in each of them.

12-15 The first available stack h together with its identifier d are extracted and removed from the queue $\bar{s}$, then its first instruction $inst$ is retrieved from h.

16-225 The extracted instruction $inst$ is matched with the available ones in order to determine what operations have to be done.

17-23 If the current instruction $inst$ is a $\langle \textbf{PROC}, T, \iota, \bar{\alpha}, \bar{\beta}, \bar{\xi}, a \rangle$ node some data structures have to be initialized. In particular, the declared input and output streams are created in the store σ as empty lists ε, then an **HALT** instruction is pushed on the current stack h, followed by the node body a. Each **HALT** instruction contains a reference to the parent stack, which in this case is ε. Finally, the stack h is added to the queue $\bar{z}$.

24-26 When an **HALT** instruction is reached, the stack can be considered empty. The interpreter simply pushes again the same **HALT** instruction on the current stack h, and adds h to the updated stack queue $\bar{z}$.

27-33 If $inst$ is a $\langle \textbf{LET}, \bar{v}, \bar{e}, a \rangle$ node, the specified variables $\bar{v}$ are stored in σ and initialized with the value of the corresponding expression in $\bar{e}$. Finally, the body a is pushed on the current stack h and this one is added to the updated stack queue $\bar{z}$.

34-37 In case the current instruction $inst$ is a $\langle \textbf{SEQ}, t, a, b \rangle$ node, its two children a and b are pushed in the current stack h using an order such that a is extracted before b, then the current stack is added to the queue $\bar{z}$.

38-44 When $inst$ is a $\langle \textbf{CHOICE}, t, \varphi, a, b \rangle$ node, the condition φ is firstly evaluated, then if its value is true child a is pushed on the current stack h, otherwise child b is pushed on h. Finally, h is enqueued to $\bar{z}$.

45-48 If the current instruction $inst$ is a $\langle \textbf{LOOP}, t, \varphi, a, b \rangle$ child a has to be executed, followed by the evaluation of the condition φ, and eventually the execution

of the other children b together with another loop iteration. More specifically, first of all a new **CHOICE** node is pushed on the current stack h with the same condition φ, the first child is a sequence composed of the other child b and the loop itself, while the second child is a $\langle$**SKIP**$\rangle$ node. Subsequently, child a is also pushed on h, and finally this stack is added to the queue $\bar{z}$.

Listing 5.2 NestFlow interpreter (2/8): classical sequential instructions.

```
        ...continue...
34          item ⟨SEQ, t, a, b⟩ do
35              h ← push(h, b);
36              h ← push(h, a);
37              z̄ ← enqueue(z̄, ⟨d, h⟩);
38          item ⟨CHOICE, t, φ, a, b⟩ do
39              if eval(σ, φ) then
40                  h ← push(h, a);
41              else
42                  h ← push(h, b);
43              end if
44              z̄ ← enqueue(z̄, ⟨d, h⟩);
45          item ⟨LOOP, t, φ, a, b⟩ do
46              h ← push(h, ⟨CHOICE, ε, φ, ⟨SEQ, ε, b, inst⟩, ⟨SKIP⟩⟩);
47              h ← push(h, a);
48              z̄ ← enqueue(z̄, ⟨d, h⟩);
49      ...continue...
```

50-53 When a $\langle$**TRY**$, t, \xi, a, b\rangle$ node is encountered, a placeholder represented by the instruction $\langle$**CATCH**$, \xi, b\rangle$ is pushed on the current stack h. This placeholder delimits the scope of the **TRY** block and determines the ability of catching an exception of type ξ in case it is thrown inside a. Then, the child a is also pushed on h which is finally added to the queue $\bar{z}$.

54-55 The extraction of a $\langle$**CATCH**$, \xi, b\rangle$ node during the normal process execution does not produce any effect: it is simply removed from the stack.

56-89 In case a $\langle$**THROW**$, \xi, \bar{t}\rangle$ node is extracted, the list $\bar{t}$ of parallel branches is initially checked. If $\bar{t}$ is not empty, then the interpreter simply pushes a **THROW** command into the corresponding stacks. More specifically, the function *push-throw* : $\mathcal{Q}ueue \times \mathcal{Q}ueue \times \mathcal{X} \times \mathcal{I}$ pushes the specified exception on the stack with the given identifer by retrieving it from one of the two queues. Conversely, a $\langle$**THROW**$, \xi, \bar{t}\rangle$ command on the current stack, requires to iteratively unroll it until a $\langle$**CATCH**$, \xi, b\rangle$ instruction annotated with the same exception type ξ is found. If such **CATCH** node is found in the current stack, child b is executed. Otherwise, if the context is the main one, i.e. its identifier is ε, the exception is propagated to the run-time system. In the other case, the sibling of the current stack is searched: the sibling stack is a stack generated by the same parallel block, while a parent stack is

Listing 5.3 NestFlow interpreter (3/8): exception handling.

```
        ...continue...
50          item ⟨TRY, t, ξ, a, b⟩ do
51              h ← push(h, ⟨CATCH, ξ, b⟩);
52              h ← push(h, a);
53              z̄ ← enqueue(z̄, ⟨d, h⟩);
54          item ⟨CATCH, ξ, b⟩ do
55              z̄ ← enqueue(z̄, ⟨d, h⟩);
56          item ⟨THROW, ξ, t̄⟩ do
57              if t̄ ≠ ε then
58                  for each t ∈ t̄ do
59                      ⟨z̄, s̄⟩ ← push-throw(z̄, s̄, ξ, t);
60                  end for
61              else
62                  b ← ε;
63                  x ← ε;
64                  while type(top(h)) = HALT ∧ x ≠ ξ do
65                      if type(top(h)) = CATCH then
66                          let ⟨CATCH, x, b⟩ = top(h);
67                      else if type(top(h)) = LIMIT then
68                          let ⟨LIMIT, t, v⟩ = top(h);
69                          ⟨g, l⟩ ← ⟨t, v⟩;
70                      end if
71                      h ← pop(h);
72                  end while
73                  if x = ξ do
74                      h ← push(h, b);
75                      z̄ ← enqueue(z̄, ⟨d, h⟩);
76                  else
77                      let ⟨HALT, t⟩ = top(h);
78                      if cn = ε then
79                          throw new UnmanagedException(ξ);
80                      else if ¬ has-sibling(z̄, s̄, d, t) then
81                          ⟨z̄, s̄⟩ ← push-throw(z̄, s̄, ξ, t);
82                      else if type(top(sibling(z̄, s̄, d, t))) = HALT then
83                          ⟨z̄, s̄⟩ ← push-throw(z̄, s̄, ParallelEx, t);
84                      else
85                          k ← sibling(z̄, s̄, d, t);
86                          ⟨z̄, s̄⟩ ← push-throw(z̄, s̄, InterruptEx, k);
87                      end if
88                  end if
89              end if
        ...continue...
```

the stack containing the parallel block generating the stack. The function *has-sibling* : $\mathcal{Q}ueue \times \mathcal{Q}ueue \times \mathcal{I} \times \mathcal{I} \to \mathbb{B}$ returns true if the stack with the given identifier and the given parent identifier has a sibling in one of the given queues, false otherwise. The parent identifier is retrieved through the HALT

node. If the current stack has no sibling, the exception is propagated to its parent. Conversely, if the current stack has a sibling and it is completed, a ParallelEx is propagated to the parent stack, while if the sibling is not completed, it is reverted by pushing on it a InterruptEx. The function $sibling : \mathcal{Q}ueue \times \mathcal{Q}ueue \times \mathcal{I} \times \mathcal{I} \rightarrow \mathcal{S}tack$ searches the sibling of the stack with the given identifier and parent identifer considering the two queues.

Listing 5.4 NestFlow interpreter (4/8): static parallel execution.

```
        ...continue...
90          item ⟨PAR, t, u, w, ψ, a, b⟩ do
91              f ← push(ε, ⟨HALT, t⟩);
92              z̄ ← enqueue(z̄, ⟨u, push(f, a)⟩);
93              z̄ ← enqueue(z̄, ⟨w, push(f, b)⟩);
94              z̄ ← enqueue(z̄, ⟨d, push(h, ⟨PAR-JOIN, ψ, u, w⟩)⟩);
95          item ⟨PAR-JOIN, ψ, u, w⟩ do
96              ⟨z̄, f⟩ ← extract(z̄, u);
97              ⟨z̄, g⟩ ← extract(z̄, w);
98              if type(top(f)) = HALT ∧ type(top(g)) = HALT then
99                  continue
100             else if type(top(f)) = HALT ∧ partial-eval(σ, ψ) then
101                 z̄ ← enqueue(z̄, ⟨u, f⟩);
102                 z̄ ← enqueue(z̄, ⟨w, push(g, ⟨THROW, InterruptEx, ε⟩)⟩);
103             else if type(top(g)) = HALT ∧ partial-eval(σ, ψ) then
104                 z̄ ← enqueue(z̄, ⟨w, g⟩);
105                 z̄ ← enqueue(z̄, ⟨u, push(f, ⟨THROW, InterruptEx, ε⟩)⟩);
106             else
107                 z̄ ← enqueue(z̄, ⟨u, f⟩);
108                 z̄ ← enqueue(z̄, ⟨w, g⟩);
109                 h ← push(h, inst);
110             end if
111             z̄ ← enqueue(z̄, ⟨d, h⟩);
        ...continue...
```

90-94 In case the current instruction *inst* is a $\langle \textsf{PAR}, t, u, w, \psi, a, b \rangle$ node, two new stacks with identifier u and w are created which contain beside to the HALT instruction, the child node a and b, respectively. Moreover, a PAR-JOIN instruction is pushed on the current stack in order to wait for the completion of the two parallel branches before continuing the execution. All the three stacks are added to the queue $\bar{z}$.

95-111 When a node $\langle \textsf{PAR-JOIN}, \psi, u, w \rangle$ is encountered, the interpreter has to determine if the execution of a related parallel branches is concluded. Therefore, it firstly retrieves the two corresponding stacks by using the function $extract : \mathcal{Q}ueue \times \mathcal{I} \rightarrow \mathcal{S}tack$ which removes from the given queue the stack with the specified indentifier, and returns such stack together with the updated queue. If both stacks are empty, namely they contain only the HALT

instruction, then the execution can continue. Otherwise, if at least one of the two branches has completed and the join condition evaluates to true using the undefined semantics, an InterruptEx is pushed on the other branch and the two stacks are added to the queue $\bar{z}$. Function *partial-eval* : $\Sigma \times \mathcal{E} \to \mathbb{B}$ evaluates the given expression using the current store content and considering the three values logic of Kleene [115, 116], where the third value is used to manage such variables that are used in the expression but are not available because still in use inside a parallel branch. If none of the previous cases if reached, the **PAR-JOIN** instruction is added again to the current stack, and the two child stacks are added to the queue $\bar{z}$. The current stack h is added to $\bar{z}$ in any case.

Listing 5.5 NestFlow interpreter (5/8): dynamic process creation.

```
        ...continue...
112         item ⟨CONCUR, t, a⟩ do
113             h ← push(h, ⟨CONCUR-JOIN, t⟩);
114             h ← push(h, a);
115             z̄ ← enqueue(z̄, ⟨d, h⟩);
116         item ⟨SPAWN, e, T, t⟩ do
117             x ← gen-id(cn);
118             q ← enqueue(q, context(cn.t.x, model(p, T), ⟨g, l⟩));
119             σ ← update-store(σ, eval(σ, e), x);
120             z̄ ← enqueue(z̄, ⟨d, h⟩);
121         item ⟨CONCUR-JOIN, t⟩ do
122             C ← find-all(q, cn.t);
123             j ← 0
124             for each c ∈ C do
125                 if halt(c) then
126                     q ← remove(q, c);
127                     j ← j + 1;
128                 end if
129             end for
130             if j < |C| then
131                 h ← push(h, inst);
132             end if
133             z̄ ← enqueue(z̄, ⟨d, h⟩);
        ...continue...
```

112-115 The execution of a $\langle$**CONCUR**$, t, a\rangle$ node initially pushes on the current stack a placeholder instruction $\langle$**CONCUR-JOIN**$, t\rangle$ that delimits the block scope, then the body a is pushed on h, and finally h is added to the queue $\bar{z}$.

116-120 The instruction $\langle$**SPAWN**$, e, T, t\rangle$ creates a new process instance of type T and stores it inside the location identified by the expression e. In particular, a new identifier is generated by function *gen-id* : $\mathcal{C}xt \to \mathcal{I}$; then, a context is created for the new process instance which is stored in the queue q. Finally,

the store is updated by assigning the identifier of the new process instance to the location given by the expression e, while the stack h is added to $\bar{z}$.

121-133 The instruction $\langle$**CONCUR-JOIN**$, t\rangle$ waits for the completion of all stacks dynamically generated inside a concur block with identifier t. In particular, the function *find-all* : $\mathcal{Q}ueue \times \mathcal{I} \to \wp(\mathcal{I})$ retrieves all the contexts in the queue q with an identifier containing the given prefix. The way identifiers are built ensures that all contexts created inside the same threshold block share the same prefix. If one of these contexts has completed, it is removed from q. The funciton *remove* : $\mathcal{Q}ueue \times \mathcal{I} \to \mathcal{Q}ueue$ removes a given the context with the given identifier from the specified queue, independently from its position. If not all contexts have been removed, the **CONCUR-JOIN** instruction is pushed again on the current stack h which is finally added to the queue $\bar{z}$.

Listing 5.6 NestFlow interpreter (6/8): static process instances and threads.

```
        ...continue...
134         item ⟨THRES, t, v, a⟩ do
135             h ← push(h, ⟨LIMIT, g, l⟩);
136             h ← push(h, a));
137             h ← push(h, ⟨LIMIT, t, σ(v)⟩);
138             z̄ ← enqueue(z̄, ⟨d, h⟩);
139         item ⟨LIMIT, t, v⟩ do
140             ⟨g, l⟩ ← ⟨t, v⟩;
141         item ⟨RUN, t, T, r̄, ō⟩ do
142             if find(q, cn.t) = ε then
143                 if ¬ is-native(T) then
144                     q ← enqueue(q, context(cn.t, model(p, T), ⟨g, l⟩));
145                 else
146                     // Create native process instance.
147                 end if
148                 h ← push(h, ⟨WAIT, cn.t⟩);
149                 z̄ ← enqueue(z̄, ⟨d, h⟩);
150             item ⟨WAIT, id⟩ do
151                 if ¬ halt(find(q, id)) then
152                     h ← push(h, inst);
153                 end if
154                 z̄ ← enqueue(z̄, ⟨d, h⟩);
155             item ⟨SKIP⟩ do
156                 z̄ ← enqueue(z̄, ⟨d, h⟩);
        ...continue...
```

134-138 The execution of a $\langle$**THRES**$, t, v, a\rangle$ node has to create a new context group with an updated threshold limit. In particular, a **LIMIT** instruction is firstly pushed on the current stack in order to identify the end of the threshold block scope, then the body a is added, and finally a new **LIMIT** instruction

is pushed which creates a new group with identifier t and a limit obtained by evaluating the variable v. Finally, the stack h is added to $\bar{z}$.

139-140 The $\langle \textsf{LIMIT}, v \rangle$ instruction simply updates the group and thread limit for the current context.

141-149 The execution of a $\langle \textsf{RUN}, t, T, \bar{r}, \bar{o} \rangle$ instruction initially determines if a context already exists for the process with identifier t, otherwise a new context is created for it. If the process is native, i.e. defined with the underlying implementation language, it is managed by the run-time support. The execution of native processes is further discusses here, it is only assumed the existence of a function $\textit{is-native} : \mathcal{T} \to \mathbb{B}$ that returns true if the model is a native one, false otherwise. Subsequently, a **WAIT** instruction is pushed on the stack h in order to wait for the completion of such context before proceeding. Finally, the stack h is added to the queue $\bar{z}$.

Listing 5.7 NestFlow interpreter (7/8): send command.

```
        ...continue...
157         item ⟨SEND, t, v̄, β̄⟩ do
158             let v̄ = ⟨v_i⟩_{i=1}^n, β̄ = ⟨β_j⟩_{j=1}^m;
159             i ← 1;
160             while i ≤ n ∧ σ(v_i) ≠ unbound do
161                 i ← i + 1;
162             end while
163             if i = n do
164                 for j ← 1 to n do
165                     ⟨x, T⟩ ← process-instance(β_j);
166                     if ¬ contains(q, cn.x) then
167                         q ← enqueue(q, context(cn.x, model(p, T), ⟨g, l⟩));
168                     end if
169                     if x = ς then
170                         σ ← update-store(σ, β_j, enqueue(σ(β_j), σ(v_j)));
171                     else
172                         ⟨_, _, δ, _, _⟩ ← find(q, cn.x);
173                         δ ← update-store(δ, β_j, enqueue(δ(β_j), σ(v_j)));
174                         q ← replace-store(q, cn.x, δ);
175                     end if
176                 end for
177             else
178                 h ← push(h, ⟨THROW, UnboundEx, ε⟩);
179             end if
180             z̄ ← enqueue(z̄, ⟨d, h⟩);
        ...continue...
```

150-154 The $\langle \textsf{WAIT}, id \rangle$ instruction waits for the completion of the context with the given identifier. In particular, it retrieves from q the context with identifier id and determines through the function *halt* if the context execution

Listing 5.8 NestFlow interpreter (8/8): receive and empty commands.

```
            ...continue...
181             item ⟨RECEIVE, t, v̄, ᾱ, θ⟩ do
182                 if time() − σ(t) > θ then
183                     h ← push(h, ⟨THROW, TimeoutEx, ε⟩);
184                     σ ← update-store(σ, t, ω);
185                 else
186                     let v̄ = ⟨v_i⟩_{i=1}^n, ᾱ = ⟨α_j⟩_{j=1}^m;
187                     k ← ω;
188                     j ← 0;
189                     for i ← 1 to n do
190                         if ¬ is-empty(σ(α_i)) ∧ ts(head(α_i)) < k then
191                             j ← i;
192                             k ← ts(head(α_i));
193                         end if
194                     end for
195                     if j > 0 then
196                         for i ← 1 to n do
197                             σ ← update-store(σ, v_i, unbound));
198                         end for
199                         ⟨x, T⟩ ← process-instance(α_j);
200                         if x = ς then
201                             σ ← update-store(σ, v_j, head(σ(α_j)));
202                             σ ← update-store(σ, σ(α_j), tail(σ(α_j)));
203                         else if contains(q, cn.x) then
204                             ⟨_, _, δ, _, _⟩ ← find(q, cn.x);
205                             σ ← update-store(σ, v_j, head(δ(α_j)));
206                             δ ← update-store(ρ, δ(α_j), tail(δ(α_j)));
207                             q ← replace-store(q, cn.x, δ);
208                         end if
209                     else
210                         h ← push(h, inst);
211                         σ ← update-store(σ, t, time());
212                     end if
213                 end if
214                 z̄ ← enqueue(z̄, ⟨d, h⟩);
215             item ⟨EMPTY, α⟩ do
216                 ⟨x, T⟩ ← process-instance(α_j);
217                 if x = ς then
218                     σ ← update-store(σ, α, ε);
219                 else if contains(q, cn.x) then
220                     ⟨_, _, δ, _, _⟩ ← find(q, cn.x);
221                     δ ← update-store(δ, α, ε);
222                     q ← replace-store(q, cn.x, δ);
223                 end if
224                 z̄ ← enqueue(z̄, ⟨d, h⟩);
225         end match
226     end while
227     q ← enqueue(q, ⟨cn, p, σ, z̄, l⟩);
228 end while
```

has been completed, otherwise the **WAIT** instruction is pushed again on the current stack h which is finally added to $\bar{z}$.

155-156 A **SKIP** instruction does not perform anything, it simply adds the current stack h to the queue $\bar{z}$.

157-180 The execution of a $\langle \textbf{SEND}, t, \bar{v}, \bar{\beta} \rangle$ instruction firstly verifies that all the specified variables are bound, otherwise a UnboundEx is thrown on the current stack. Subsequently, the identifier and the type of the process instance associated to each stream β_i is determined through the function $\mathit{process\text{-}instance} : \mathcal{I} \to \mathcal{I} \times \mathcal{T}$. It is assumed that the name of each stream is fully qualified, allowing one to retrieve its corresponding process instance. If the process instance associated to the stream is equal to this, then the current store is updated by adding the variable value to the stream. Otherwise, the context of the involved process instance is retrieved through function $\mathit{find} : \mathcal{Q}\mathit{ueue} \times \mathcal{I} \to \mathcal{C}\mathit{xt}$ and the corresponding store is then updated. Function $\mathit{replace\text{-}store} : \mathcal{Q}\mathit{ueue} \times \mathcal{I} \times \Sigma \to \mathcal{Q}\mathit{ueue}$ updates the provided queue of contexts by substituting the store associated to the context identifier with the new one. Finally, stack h is added to $\bar{z}$.

181-214 The execution of a $\langle \textbf{RECEIVE}, t, \bar{v}, \bar{\alpha}, \theta \rangle$ firstly determines if the specified timeout θ is elapsed: in this case a TimeoutEx is thrown and the store is updated by resetting the timestamp associated to the receive. Procedure $\mathit{time}()$ returns the current time. Otherwise, a not empty stream is selected using a fair strategy; in particular, function $ts : \mathcal{U} \to \mathbb{R}$ returns the timestamp associated to the given object and is used to choose the first arrived object. If at least a not empty stream is found all variables are reset to unbound, while the one corresponding to the chosen stream will contain the extracted object. If the chosen stream belongs to the current process instance, the store σ is updated, otherwise the context of the corresponding process instance is retrieve and its store δ is updated. The queue associated to the corresponding stream is also updated. Finally, stack h is added to $\bar{z}$.

215-224 The empty command simply empties the queue associated to the given stream. In particular, if the stream does not belong to the current process instance, its corresponding context is firstly retrieved. At the end stack h is added to $\bar{z}$.

227 The final operation performed by the cycle is added the updated context to queue q.

◊

5.10 Summary and Concluding Remarks

This chapter promotes the adoption of a structured approach to process modeling, in place of an unstructured one, in order to reduce the presence of control-flow errors inside a model and at the same time to increase its modularity and comprehensibility. The chapter starts by analysing several reasons that justify the adoption of a structured approach. In particular, the use of a free-composition paradigm does not prevent the introduction of subtle errors that are difficult to detect and recover even using sophisticated verification methods. Moreover, many available PMLs can make recurring patterns hard to recognize, since exactly the same logic can be displayed in very different ways with no additional efforts. Conversely, using a structured approach, the only way to change the representation of a model is expressing the same logic in a very different way. Additionally, the modularity of an unstructured PML can be very low, hindering its applicability in the design of complex real processes. Finally, many arguments used to neglect a structured modeling approach are proven to be ill-founded and not applicable in the general case. This justify the introduction of a novel PML, called NestFlow, at least for exposing a different design solution that takes structure and modularity in serious consideration. NestFlow combines structured control-flow constructs with AMP abstractions exploiting the following ideas: stateful instead of stateless components, nested control-flow structures, unique correspondence between component instances and graphical occurrences, no shared variables among parallel entities and data-flow subsumption. Its principal aim is to discuss how a structured PML can be effectively built, not to provide a fully-fledged system ready to use, even if its core version can be the ideal starting point. The remainder of the chapter introduces the NestFlow constructs and its formal semantics, while its expressiveness and some of its applications are discussed in the following chapter.

6

NestFlow Expressiveness and Applications

The previous chapter has introduced NestFlow, a novel structured PML for modular process design. NestFlow has been conceived as a proof-of-concept language, rather than a fully-fledged system: its aim is to prove that a PML with a structured control-flow can be realized providing both comprehensibility and modularity thanks to component encapsulation and AMP interfaces. This chapter starts by discussing its expressiveness and suitability in the PAIS domain using the well-known workflow control-flow patterns. This method has been widely adopted in literature for evaluating several different offerings [32–35] and it provides a good starting point for PML comparison. Anyway, the used evaluation method is sometimes criticized [36] because it is only based on the analysis of the available constructs; therefore, a more objective approach is proposed here which consider the effort needed to replicate the behaviour described by each pattern. The performed evaluation will demonstrate that the behaviour prescribed by the various patterns can be obtained in NestFlow using a small set of constructs compared with the reference CPNs implementation given in [12].

Besides to the business process management domain, NestFlow may be successfully applied in other contexts. In particular, it has been used in the geographical domain for modeling long-running interactive computations on huge amount of data. As discussed in [22] this domain can benefit from the adoption of a data-flow modeling language enhanced with coarse-grained control-flow constructs to compactly express the emerging process logics. Another explored application regards the modeling of clinical processes in which several temporal constraints have to be expressed for guaranteeing a successful process execution. For this purpose, a variant of NestFlow, called Tnest, has been introduced in [23] which provides the ability to specify temporal constraints on single tasks and among tasks, and to perform some controllability checks on the modeled processes.

The remainder of this chapter is organized as follows: Sec. 6.1 evaluates the expressiveness and suitability of NestFlow for business process design using the workflow control-flow patterns methodology. Sec. 6.2 discusses the application of NestFlow for the design of geographical processes, while Sec. 6.3 presents the Tnest extension which allows the specification of temporal constraints of particular interest in the health-care domain.

6.1 Workflow Management

In the business process management community, the workflow pattern initiative [12] provides a framework to evaluate the suitability of WfMSs for business process design based on a set of recurring features, called *patterns*. In literature many offerings have been evaluated against workflow patterns [32–35], providing a good starting point for system comparison. Inspired by these contributions, this section evaluates the suitability and expressiveness of NestFlow for business process design in terms of supported Workflow Control-Flow Patterns (WCPs) [12,15]. The aim is to give an idea on how a structured PML can represent the behavior of most WCPs with a small set of control-flow and data-flow constructs.

WCPs share many commonalities and a pattern-based evaluation that does not consider these relations will be less usable and mostly redundant. For this reason, a compact classification of WCPs is introduced which groups patterns by similarity in four generic Workflow Control-Flow *Parametric* Patterns (WCPPs): Wcpp-Seq, Wcpp-Cancel, Wcpp-Fork and Wcpp-Synch. Repetition (G2), Trigger (G4) and Termination (G5) Patterns are excluded from this classification, because for them differences and commonalities are trivial. Each WCPP will be presented inside the section of the corresponding pattern group through a table, which explains how a particular WCP can be obtained fixing the parameters. For each parameter the table reports when it is known, at design-time (DT), at run-time before activation (RT), or at run-time after activation (FT), and its value separated by a slash.

6.1.1 Sequence Patterns (G1)

Group G1 captures patterns involved the specification of the task execution order, such as: Sequence Wcp-01, Interleaved P.O. Routing Wcp-17, Interleaved Routing Wcp-40, Critical Section Wcp-39, and Milestone Wcp-18. They can be obtained from the parametric pattern Wcpp-Seq(S, R) as shown in Tab. 6.1, except for the last two: Critical Session Wcp-39 and Milestone Wcp-18.

Table 6.1. Sequence Patterns (G1) captured by Wcpp-Seq(S, R). S is the set of involved tasks and $R \subseteq S \times S$ is a partial order on S represented as a direct acyclic graph.

Pattern	$\lvert S\rvert$	S	$\lvert R\rvert$	R
Wcp-01	DT/$\lvert S\rvert = k$	DT	DT/$\lvert R\rvert = k-1$	DT/$R = SEQ^{(a)}$
Wcp-17	DT/$\lvert S\rvert = k$	DT	DT/$\lvert R\rvert \in \{1, ..., k-1\}$	DT/$R = PO^{(b)}$
Wcp-40	DT/$\lvert S\rvert = k$	DT	DT/$\lvert R\rvert = 0$	DT/$R = \varnothing$

$^{(a)}\{(x_i, x_{i+1}) \mid 1 \leq i < n\}$ $^{(b)}$Partial order on S given as a directed acyclic graph.

Wcp-01 Sequence pattern requires to define a total execution order among the involved tasks, while Wcp-17 Interleaved P.O. Routing prescribes that only one task at time can be executed but following a partial ordering among them, finally Wcp-40 Interleaved Routing is similar to the previous one but no ordering is required among the selected tasks. In Tab. 6.1 the symbol T denotes the set

of all tasks contained in the model, $S \subseteq T$ is the subset of tasks involved in the pattern and $R \subseteq S \times S$ is an ordering relation on S. The set S is known at design time, as well as the required total, partial, or empty ordering relation.

WCP-01 Sequence

Description – A task in a process is enabled after the completion of a preceding activity in the same process [12].

Realization – Sequence (WCP-01) can be obtained in NESTFLOW through one or more sequence block, as depicted in Fig. 6.1. □

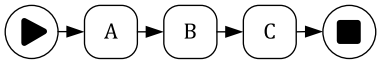


Fig. 6.1. WCP-01 Sequence implementation in NESTFLOW: task B is enabled after the completion of task A, while task C is enabled after the completion of B.

WCP-17 Interleaved P.O. Routing

Description – A set of tasks has a partial ordering defining the order in which they must be executed. Each task in the set must be executed once and they can by completed in any order according with the partial order. However, as an additional requirement, no two tasks can be executed at the same time (i.e. no two tasks can be active for the same process instance at the same time) [12].

Realization – Any finite partial order with mutual exclusion $R \subseteq S \times S$ defined on the set of tasks S and represented as a direct acyclic graph, can be obtained in NESTFLOW using a parallel block, sequence blocks and link constructs, surrounded by a threshold block with $k = 1$. The surrounded threshold block ensures that at any time all enabled tasks are offered, but when k of these is chosen the other ones are suspended.

A naive construction can be obtained with a single parallel block containing exactly one parallel branch for each task instance $t \in S$, then for each relation $(u, v) \in R$, a send s is added at the end of the branch containing u, a receive r is added at the beginning of the branch containing v and a link is added between s and r, as for A and B in Fig. 6.2.a. The threshold block with $k = 1$ ensures that only one thread of control at time can execute inside the parallel block; hence, only one task at time is running, while the other ones are suspended. NESTFLOW supports even more general situations, because k can be greater than one and, in a more sophisticated implementation, may vary at run-time. This generic construction is used to prove the support of Interleaved P.O. Routing (WCP-17), but simpler constructions are possible, as exemplified by Fig. 6.2.b which is equivalent to the fragment in Fig. 6.2.a. □

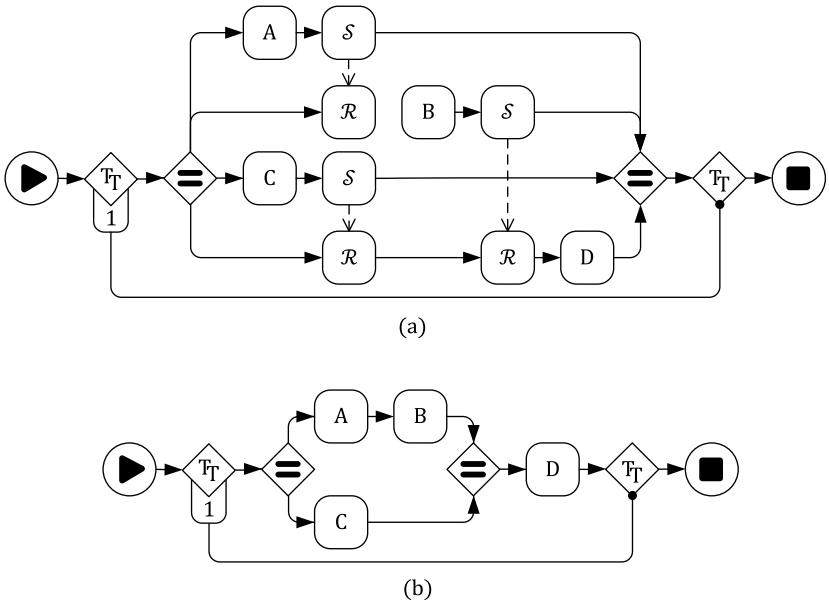


Fig. 6.2. (a) and (b) present two possible realizations of Interleaved P.O. Parallel Routing (Wcp-17) for a set of four tasks with the same dependency relationships represented by the partial order $R = \{(A, B), (B, D), (C, D)\}$.

Wcp-40 Interleaved Routing

Description – Each member of a set of tasks must be executed once. They can be executed in any order but no two tasks can be executed at the same time (i.e. no two tasks can be active for the same process instance at the same time). Once all of the tasks have completed, the next task in the process can be initiated [12].

Realization – Interleaved Routing (Wcp-40) is a specialization of Interleaved P.O. Routing (Wcp-17) where the partial order $R = \varnothing$. Therefore, it can be implemented

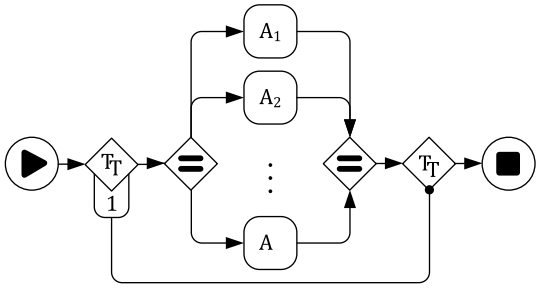


Fig. 6.3. Wcp-40 Interleaved Routing implementation in NestFlow.

by distributing set of tasks $S = \{A_1, \ldots, A_n\}$ into a parallel block wrapped by a threshold block with $k = 1$, as depicted in Fig. 6.3. This block ensures that only

one thread of control at time can execute inside the parallel block, hence, only one task at time is running, while the other ones are suspended. As mentioned for the previous pattern, NestFlow supports even more general situations, because k can be greater than one and, in a more sophisticated implementation, may vary at run-time. □

Wcp-18 Milestone

Description – A task is only enabled when the process instance (of which it is part) is in a specific state (typically a parallel branch). The state is assumed to be a specific execution point (also known as a milestone) in the process model. When this execution point is reached, the nominated task can be enabled. If the process instance has progressed beyond this state, then the task cannot be enabled now or at any future time (i.e. the deadline has expired). Note that the execution does not influence the state itself, i.e. unlike normal control-flow dependencies it is a test rather than a trigger [12].

Realization – In Milestone (Wcp-18) a task B can execute only when the process instance is in a specific state, for example another task A is just concluded. This pattern can be represented as in Fig. 6.4: after the completion of A, an object is sent to the branch containing B, if the thread of control in this branch is blocked in the receive command, namely it is waiting that a specific state is reached, then the object wakes up the thread suspended on the receive and B is executed; otherwise, if the receive is performed some time later the completion of A, the empty command deletes all objects previously received in the stream $r.y_{in}$ associated to the variable y and B is not executed. □

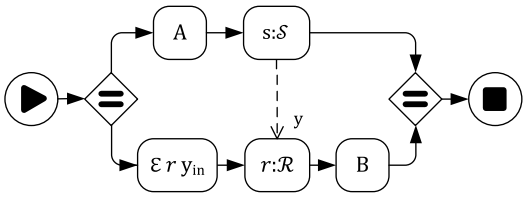


Fig. 6.4. Wcp-18 Milestone implementation in NestFlow.

Wcp-39 Critical Section

Description – Two or more connected subgraphs of a process model are identified as "critical sections". At runtime for a given process instance, only tasks in one of these "critical sections" can be active at any given time. Once execution of the tasks in one "critical section" commences, it must complete before another "critical section" can commence [12]. *Realization* – Critical Section (Wcp-39) assumes the

presence of a shared resource that has to be accessed in a mutually exclusive way. This resource can be managed by a single task instance and the other tasks can

gain access to the underlying resource only sending objects to it. For example, the resource can be a log file, while the sent objects can be the entries to be appended to this shared file. Stream serialization ensures the exclusive access to the resource. This solution is depicted in Fig. 6.5.a, where task A manages the shared resource: it receives the requests of the other tasks and sends back to them some information if necessary.

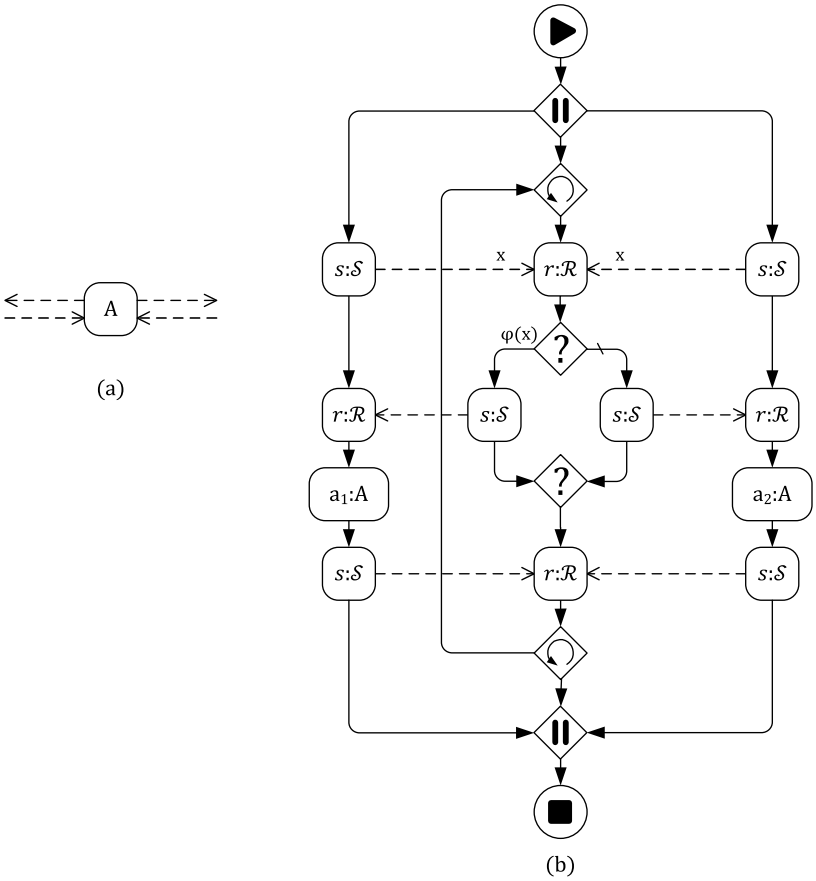


Fig. 6.5. WCP-39 Critical Section implementation in NestFlow. (a) The simplest implementation with a single task instance. (b) Representation with a unique object exchanged through streams.

A shared resource can also be represented with a unique object exchanged by tasks using links, as in Fig. 6.5.b. Tasks $a_1 : A$ and $a_2 : A$ need the shared resource to execute: before starting they send a request for the resource and wait to receive it. Only one request at time is satisfied, hence only one task between $a_1 : A$ and $a_2 : A$ can execute at a certain time. Notice that in the example of Fig. 6.5.b the discrimination between the two tasks is made on the basis of the value received in x, alternatively one can use a distinct variable for each involved task and check

which variable is not unbound. After termination, the task that currently holds the resource, send a message to notify its completion and unlock the resource.

Anyway, the pattern WCP-39 has been conceived for those languages that allow the presence of a shared state, this is not the case of NESTFLOW; hence, the given implementation is intended for managing external shared states. □

6.1.2 Repetition Patterns (G2)

Repetition patterns describe various ways in which repetitive tasks or sub-processes can be specified in a process [15]. This group includes: Arbitrary Cycles WCP-10, Structured Loop WCP-21, and Recursion WCP-22.

WCP-10 Arbitrary Cycles

Description – The ability to represent cycles in a process model that have more than one entry or exit point. It must be possible for individual entry and exit points to be associated with distinct branches [12].

Realization – Arbitrary cycles (WCP-10) are avoided in NESTFLOW, because they drastically reduce modularity: this is not a severe limitation since any sequential composition of nested arbitrary cycles has an equivalent structured form [117]. □

WCP-21 Structured Loop

Description – The ability to execute a task or sub-process repeatedly. The loop has either a pre-test or post-test condition associated with it that is either evaluated at the beginning or end of the loop to determine whether it should continue. The looping structure has a single entry and exit point [12].

Realization – Structured Loop (WCP-21) is directly supported in NESTFLOW through the loop block, as depicted in Fig. 6.6.a. This can also represent while-do and repeat-until blocks by placing a skip in the left or right branch, as in Fig. 6.6.b and Fig. 6.6.c, respectively. □

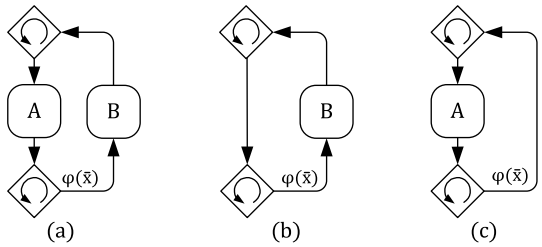


Fig. 6.6. WCP-21 Structured Loop implementation in NESTFLOW. (a) Generic structured loop, (b) while-do loop and (c) repeat-until loop.

Wcp-22 Recursion

Description – The ability of a task to invoke itself during its execution or an ancestor in terms of the overall decomposition structure with which it is associated [12].

Realization – Recursive declarations of tasks (Wcp-22) are supported in NestFlow through lazy evaluation. The model in Fig. 6.7 shows an example of recursive declaration of a task that computes the *x-th* Fibonacci number. In this example a compact notation is used for specifying the variables read and written by the inner task *fib*. In particular, the variables above the task name are the read variables, while the variables below the task name are the written ones. Notice that this example allows parallel branches to read the value of the same variable x; anyway, it is always forbidden that a variable appears as a left-value in assignments of different parallel branches. Alternatively,the assignments can be moved $u \leftarrow x - 1$ and $v \leftarrow x - 2$ just before the parallel block. □

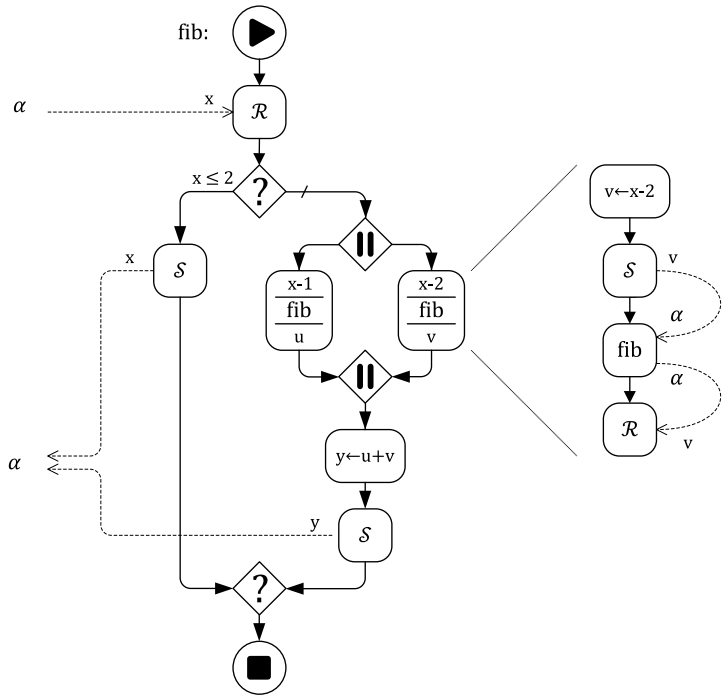


Fig. 6.7. Example of recursive declaration (Wcp-22) of a task that computes the *x-th* Fibonacci number.

6.1.3 Cancellation Patterns (G3)

Cancellation Patterns represents different ways through which one or more tasks can be withdrawn. This group includes: Cancel Activity WCP-19, Cancel Case WCP-20, and Cancel Region WCP-25. These patterns are captured by the parametric pattern WCPP-CANCEL(S) described in Tab. 6.2, by fixing the set S of tasks to be cancelled. The symbol T denotes the sets of all tasks in the model. In particular WCP-19 Cancel Activity requires to cancel a single task x, while WCP-25 Cancel Region regards the cancellation of a group of tasks $S \subseteq T$, and WCP-20 Cancel Case requires to cancel all tasks of an entire case.

Table 6.2. Cancellation Patterns (G3) captured by WCPP-CANCEL(S). $S \subseteq T$ is the set of task to be cancelled and the element x is any single task in T.

Pattern	$\|S\|$	S
WCP-19	DT/$\|S\| = 1$	DT/$S = \{x\}$
WCP-25	DT/$\|S\| < \|T\|$	DT/$S \subseteq T$
WCP-20	DT/$\|S\| = \|T\|$	DT/$S = T$

WCP-19 Cancel Activity

Description – An enabled activity is withdrawn prior to its commencing execution. If the activity has started, it is disabled and, where possible, the currently running instance is halted and removed [12].

Realization – Pattern WCP-19 requires only the cancellation of an enabled or scheduled task that is not yet executed. For compound tasks this description is not sufficient, because a task may be partially executed. This section extends the behavior prescribed by the pattern including also the cancellation of activities that are still executing or are partially executed.

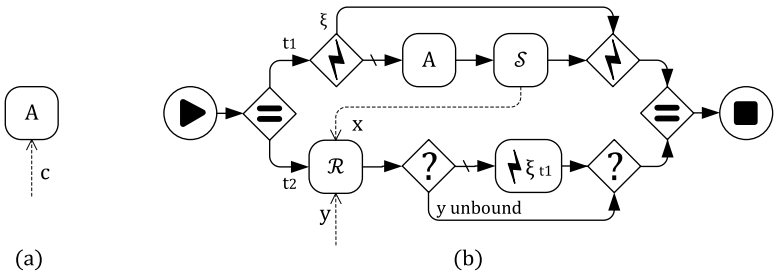


Fig. 6.8. WCP-19 Cancel Activity implementation in NESTFLOW.

NESTFLOW supports cancellation through the hierarchical exception handling constructs try and throw. Exception handling provides not only a mechanism to

manage task cancellation, but also the ability to specify clean-up actions to perform during the cancellation phase. The cancellation of a task from a different thread of control can be simply obtained by sending a cancellation message to it, as in Fig. 6.8.a. If a cancellation signal is sent to a task before it is enabled, the task may decide to consider or not the cancellation request.

In order to ensure encapsulation, each task has to be built cancellable, namely able to manage cancellation signals, because only the task knows exactly how to exit. Anyway, when a cancellation facility is not available, the exception handling mechanism can be used. A task A that does not offer a cancellation interface can be wrapped in the structure of Fig. 6.8.b: if a cancellation signal is sent to $\mathcal{R}$ during the execution of A, an interruption exception ξ is thrown on branch t_1 and the activities in this branch are cancelled; otherwise, A sends after its completion a message to $\mathcal{R}$ for terminating branch t_2 without raising an exception. □

Wcp-20 Cancel Case

Description – A complete process instance is removed. This includes currently executing activities, those which may execute at some future time and all sub-processes. The process instance is recorded as having completed unsuccessfully [12].

Realization – Cancel Case (Wcp-20) can be obtained through hierarchical exception handling constructs try and throw, as in Fig. 6.9. In particular, A is a compound task representing the entire case, if an exception of type ξ is thrown during the execution of A, the entire case is reverted and task B is executed to properly clean-up the execution, for instance to execute compensation. □

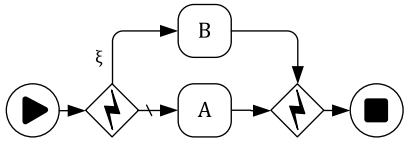


Fig. 6.9. Wcp-20 Cancel Case implementation in NestFlow.

Wcp-25 Cancel Region

Description – The ability to disable a set of activities in a process instance. If any of the activities are already executing, then they are withdrawn. The activities do not need to be a connected subset of the overall process model [12].

Realization – As stated for Cancel Activity (Wcp-19), each activity in NestFlow can be withdrawn by sending a cancellation message to it. Therefore, to cancel a set of activities, it is sufficient to send to each of them a cancellation signal, as depicted in Fig. 6.10.a where F sends a cancellation message to A, C and D. □

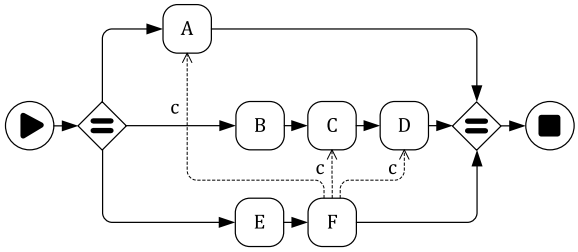


Fig. 6.10. WCP-25 Cancel Region implementation in NESTFLOW.

6.1.4 Trigger Patterns (G4)

The trigger patterns provide a means for the execution of a process to be synchronized with its broader operational environment [15]. In particular, two kinds of trigger are recognized: Transient Trigger WCP-23 and Persistent Trigger WCP-24.

WCP-23 Transient Trigger

Description – The ability for a task instance to be triggered by a signal from another part of the process or from the external environment. These triggers are transient in nature and are lost if not acted on immediately by the receiving task. A trigger can only be utilized if there is a task instance waiting for it at the time it is received [12].

Realization – The activation of a task A through an external signal is naturally supported in NESTFLOW by a link connected to A. In NESTFLOW streams are persistent, namely they retain sent objects until the receiver is able to receive them. A Transient Trigger (WCP-23) can be simulated by placing an empty command before the receive, hence all objects sent before A starts will be deleted, as depicted in Fig. 6.11 where the empty command deletes all objects contained in the stream $r.y_{in}$ associated to the variable y. □

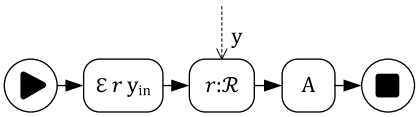


Fig. 6.11. WCP-23 Transient Trigger implementation in NESTFLOW.

WCP-24 Persistent Trigger

Description – The ability for a task to be triggered by a signal from another part of the process or from the external environment. These triggers are persistent in form and are retained by the process until they can be acted on by the receiving task [12].

Realization – The activation of a task A through an external signal is naturally supported in NestFlow by a link connected to A, as depicted in Fig. 6.12. In NestFlow streams are persistent, namely they retain sent objects until the receiver is able to receive them. □

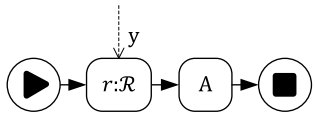


Fig. 6.12. Wcp-24 Persistent Trigger implementation in NestFlow.

6.1.5 Termination Patterns (G5)

The termination patterns identify two distinct schemes for determining when a process instance is complete [15]. This group includes: Implicit Termination Wcp-11 and Explicit Termination Wcp-43.

Wcp-11 Implicit Termination

Description – A given process (or sub-process) instance should terminate when there are no remaining work items that are able to be done either now or at any time in the future and the process instance is not in deadlock [12].

Realization – In NestFlow Implicit Termination (Wcp-11) coincides with the Explicit Termination (Wcp-43), because it offers only structured control-flow constructs. Therefore, only one thread of control can enter a block and when a thread leaves a block, no other threads remain inside it. As a result, only one thread of control reaches the stop place and the stop place is reached only when the last task instance completes. □

Wcp-43 Explicit Termination

Description – A given process (or sub-process) instance should terminate when it reaches a nominated state. Typically this is denoted by a specific end node. When this end node is reached, any remaining work in the process instance is cancelled and the overall process instance is recorded as having completed successfully, regardless of whether there are any tasks in progress or remaining to be executed [12].

Realization – Explicit Termination (Wcp-43) is problematic in concurrent context, because it may leave the case into an inconsistent state. In NestFlow consistency can be guaranteed by raising interruption exceptions in concurrent branches. Indeed, an explicit termination does not differ from exceptions thrown in concurrent executions to force completion.

Besides the stop place, in NestFlow an explicit termination can be obtained in any place of the model by raising an abort exception properly managed in concurrent branches, as in Fig. 6.13.a. It is straightforward to add a specific construct to implement such behavior, as in Fig. 6.13.b. However, even in presence of such construct, it is not simple to deal with explicit termination:for this reason such construction has been excluded from the core language. □

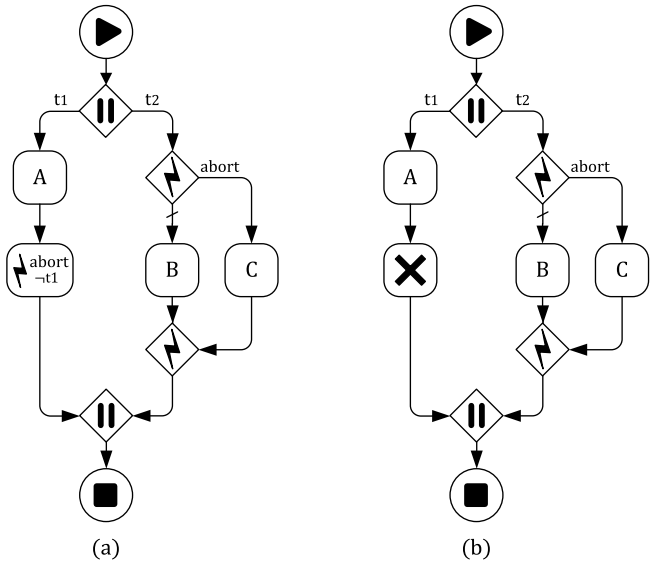


Fig. 6.13. (a) Explicit termination (Wcp-43) implemented using an interruption exception. (b) Explicit termination (Wcp-43) implemented using a specific construct.

6.1.6 Branching Patterns (G6)

Branching patterns describe the divergence of thread of controls from a single point in the model. This group includes: Parallel Split Wcp-02, Exclusive Choice Wcp-04, Multi Choice Wcp-06, Deferred Choice Wcp-16, and Thread Split Wcp-42. They can be captured by Wcpp-Fork (Q, P) in Table 6.3, where Q denotes the

Table 6.3. Branching Patterns (G6) captured by Wcpp-Fork(Q, P). Q denotes the set of involved branches and is always known at DT, $P \subseteq Q$ is the set of branches that are activated by the pattern and the element x is any branch in Q.

Pattern	$\|P\|$	P
Wcp-04	DT/$\|P\| = 1$	RT/$P = \{x\}$
Wcp-06	RT/$\|P\| < \|Q\|$	RT/$P \subseteq Q$
Wcp-02	DT/$\|P\| = \|Q\|$	DT/$P = Q$
Wcp-16	DT/$\|P\| = 1$	FT/$P = \{x\}$

set of all involved branches and $P \subseteq Q$ is the subset of branches that are activated by the pattern.

Wcp-02 Parallel Split

Description – The divergence of a branch into two or more parallel branches each of which execute concurrently [12].

Realization – Parallel Split (Wcp-02) is obtained in NestFlow with a single parallel block, as illustrated in Fig. 6.14. □

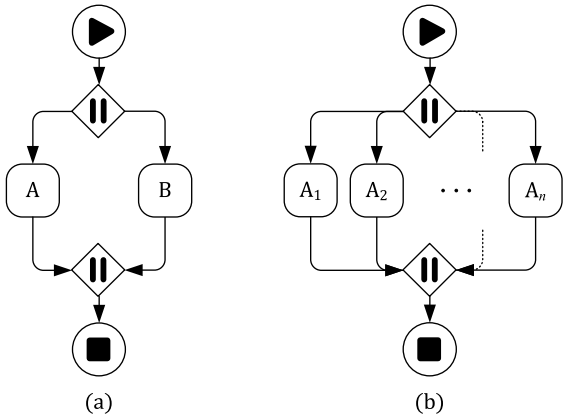


Fig. 6.14. Wcp-02 Parallel Split implementation in NestFlow. (a) The simplest case with only two tasks. (b) The general case with n tasks.

Wcp-04 Exclusive Choice

Description – The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on the outcome of a logic expression associated with the branch [12].

Realization – The Exclusive Choice (Wcp-04) behavior can be obtained in NestFlow with a single choice block, as depicted in Fig. 6.15. □

Wcp-06 Multi Choice

Description – The divergence of a branch into two or more branches. When the incoming branch is enabled, the thread of control is immediately passed to one or more of the outgoing branches based on the outcome of distinct logic expressions associated to each branch [12].

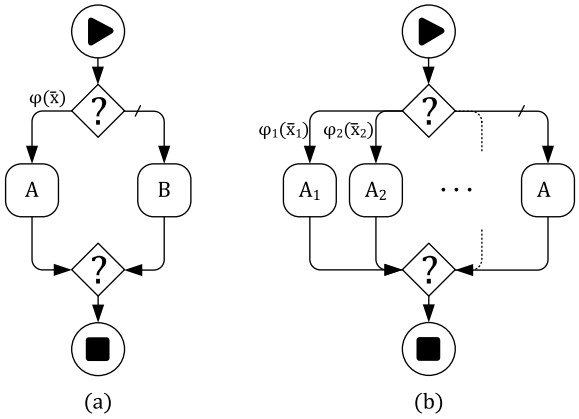


Fig. 6.15. WCP-04 Exclusive Choice implementation in NESTFLOW. (a) The simplest case with only two tasks. (b) The general case with n tasks.

Realization – There is no specific constructs for the Multi Choice (WCP-06) in NESTFLOW core language, because its behavior can be obtained combining a parallel block with one choice block for each branch. A branch has to be chosen to be the default one. For this branch the guard is the negation of the disjunction of the conditions contained in the other branches: $\varphi_n(\bar{z}) = \neg \bigvee_{i=1}^{n-1} \varphi_i(\bar{x}_i)$, where $\bar{z} = \bigcup_{i=1}^{n-1} \bar{x}_i$. This condition ensures that the default branch is enabled only when the conditions associated to the other branches are all false.

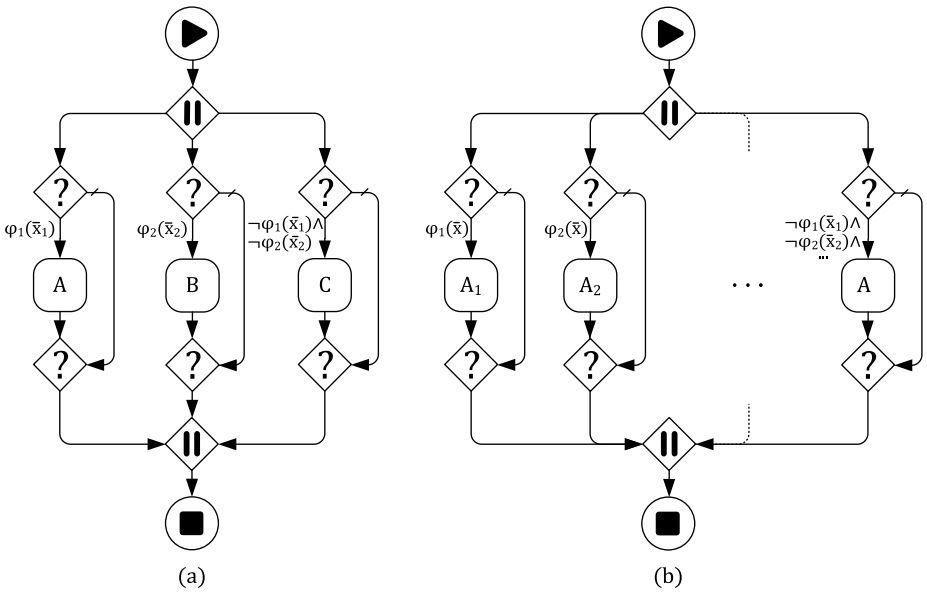


Fig. 6.16. WCP-06 Multi Choice implementation in NESTFLOW. (a) A simple case with only three tasks. (b) The general case with n tasks.

Adding a specific language construct for Wcp-06 is only an implementation issue. Fig. 6.17 depicts the pattern implementation with a specific construct. □

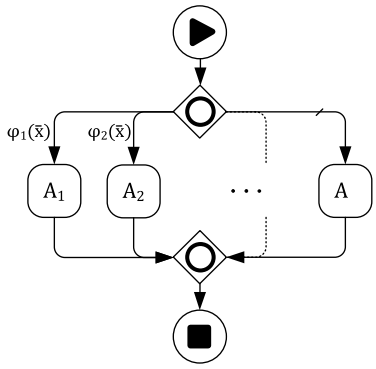


Fig. 6.17. Wcp-06 Multi Choice implementation in NestFlow with a specific construct.

Wcp-16 Deferred Choice

Description – A point in a process where one of several branches is chosen based on interaction with the operating environment. Prior to the decision, all branches represent possible future courses of execution. The decision is made by initiating the first task in one of the branches, i.e. there is no explicit choice but rather a race between different branches. After the decision is made, execution alternatives in branches other than the one selected are withdrawn [12].

Realization – Deferred Choice (Wcp-16) is the most contrived WCP because any system has its own implementation. In BPMN with WS-BPEL executable semantics it is supported by a `<pick/>` construct that allows the thread of control to be suspended, waiting for one external event chosen from a set of declared ones, for instance a received message or an expired timeout.

In YAWL the deferred choice places two or more tasks in the work-list; when one of these is chosen, the other ones are instantaneously withdrawn.

In NestFlow the first behavior can be obtained by a single receive command that waits for the first incoming object from multiple streams or for a timeout event, as in Fig. 6.18.a. The second behavior can also be obtained in NestFlow, but here a slightly different solution is preferred that is more user-friendly and is based on the suspension semantics of the threshold block. This solution is exemplified in Fig. 6.18.b, where $\neg t_i$ in the throw commands means raise an exception to interrupt all parallel branches except t_i. The threshold block ensures that only one branch is executed at a time, while the other ones are suspended. Suspended tasks are not removed from the work-list, but they cannot be chosen by users. If the chosen task completes successfully, it throws an exception of ξ on the other branches containing the suspended tasks for definitively remove them, otherwise they become available again as alternative of the failed one. In Fig. 6.18.b the task

B can also throw an exception of type c during its execution, in this case task D is executed and A becomes available again after D completion; otherwise, if B completes successfully, a ξ exception is thrown on t_1 and A is removed. □

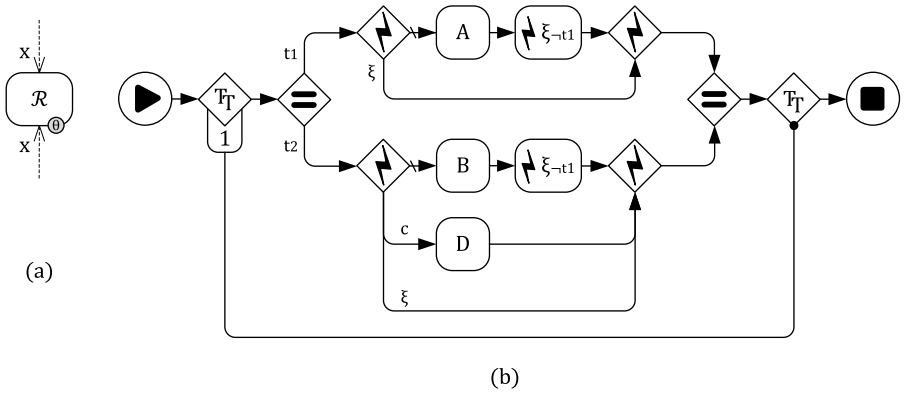


Fig. 6.18. WCP-16 Deferred Choice implementation in NESTFLOW. (a) The simplest representation of deferred choice. (b) A more sophisticated deferred choice with task suspension semantics, in case of only two alternatives. (c) An example of deferred choice with task suspension semantics and resume in case of failure.

WCP-42 Thread Split

Description – At a given point in a process, a nominated number of execution threads can be initiated in a single branch of the same process instance [12].

Realization – Thread Split (WCP-42) is not supported as a matter of principle, because it deals with explicit thread manipulation. □

6.1.7 Synchronization Patterns (G7)

Synchronization patterns capture various ways through which different thread of controls can be merged in a given point in the model. This group includes the patterns: Synchronization WCP-03, Structured Synchronizing Merge WCP-07, Structured Discriminator WCP-09, Blocking Discriminator WCP-28, Structured Partial Join WCP-30, Blocking Partial Join WCP-31, Canceling Partial Join WCP-32, Generalized And-Join WCP-33, Acyclic Synchronizing Merge WCP-37, Generalized Synchronizing Merge WCP-38, Simple Merge WCP-05, Multi Merge WCP-08, and Thread Merge WCP-41.

All synchronization patterns can be captured by the parametric pattern WCPP-SYNCH (Q, P, δ, λ) in Table 6.4, where Q denotes the set of all involved branches, $P \subseteq Q$ is the subset of branches that are synchronized by the pattern, δ is the action to be performed on the remaining $Q \setminus P$ branches and λ is the number of divergency points from which the branches in Q come from. In particular, WCP-09 Structured Discriminator requires to waits for the completion of one of the

branches in Q before continuing with the following tasks, all branches in Q have to originate from the same divergency point. WCP-28 differs from WCP-09 for the fact that the branches in Q can originate from different divergency points, while WCP-29 differs from WCP-28 for the management of the remaining branches, which are withdrawn instead of blocked at completion. WCP-30 Structured Partial Join, WCP-31 Blocking Partial Join, and WCP-32 Canceling Partial Join are equivalent to WCP-09 Structured Discriminator, WCP-28 Blocking Discriminator, and WCP-29 Canceling Discriminator, respectively, but they require the completion of $1 < m < |Q|$ branches in Q. When the number m of branches to be waited for is equal to $|Q|$, the needed pattern is WCP-03 Synchronization or WCP-33 Generalized And-Join depending on whether the branches come from the same divergency point or not, respectively. If the branches to be synchronized are all the *active* ones, the interested pattern is WCP-07 Structured Synchronizing Merge, WCP-37 Acyclic Synchronizing Merge, or WCP-38 Generalized Synchronizing Merge, depending on the presence of a structured form or the absence of cycles. Finally, when the number of branches to be waited for is one, but any other subsequent completion generates a new execution of the following tasks, the required pattern can WCP-05 Simple Merge or WCP-08 Multi Merge, where the second one allows concurrent executions of the following tasks.

Table 6.4. Synchronization Patterns (G7) captured by WCPP-SYNCH(Q, P, δ, λ). The possible values for δ are: blocking (bl), if the completion of the activities on the remaining branches has no effects, canceling (cl), if the activities on the remaining branches are cancelled, or passing (ps), if after the completion of the remaining activities the thread of control continues with the subsequent common branch. The number of divergency points λ can be 1, if all branches are originated from the same construct, or m, if the branches can be generated from one or many constructs. In column $|P|$ $1 < k < |Q|$ is the number of branches to be synchronized, while the set Q and its cardinality are always known at design-time.

Pattern	$\|P\|$	P	δ	λ
WCP-09	DT/1	RT/$P = \{q\}$	bl	1
WCP-28	DT/1	RT/$P = \{q\}$	bl	m
WCP-X1 [(a)]	DT/1	RT/$P = \{q\}$	cn	1
WCP-29	DT/1	RT/$P = \{q\}$	cn	m
WCP-30	DT/k	RT/$P \subseteq Q$	bl	1
WCP-31	DT/k	RT/$P \subseteq Q$	bl	m
WCP-X2 [(a)]	DT/k	RT/$P \subseteq Q$	cn	1
WCP-32	DT/k	RT/$P \subseteq Q$	cn	m

Pattern	$\|P\|$	P	δ	λ
WCP-07	RT/k	RT/$P \subseteq Q$	–	1
WCP-37	RT/k	RT/$P \subseteq Q$	–	m
WCP-38	RT/k	RT/$P \subseteq Q$	–	m
WCP-03	DT/$\|Q\|$	RT/$P = Q$	–	1
WCP-33	DT/$\|Q\|$	RT/$P = Q$	–	m
WCP-05	DT/1	RT/$P = \{q\}$	–	1
WCP-08	DT/1	RT/$P = \{q\}$	ps	m

[(a)] X-Patterns identify missing combinations

WCP-03 Synchronization

Description – The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled [12].

□*Realization* – Synchronization (WCP-03) can be obtained in NESTFLOW using a parallel block. Notice that NESTFLOW is a block-structured language: it does not have distinct constructs for starting and closing a parallel block; hence, the implementation of this pattern is the same of Parallel Split (WCP-02). □

WCP-07 Structured Synchronizing Merge

Description – The convergence of two or more branches (which diverged earlier in the process at a uniquely identifiable point) into a single subsequent branch. The thread of control is passed to the subsequent branch when each active incoming branch has been enabled. The Structured Synchronizing Merge occurs in a structured context, i.e. there must be a single Multi-Choice construct earlier in the process model with which the Structured Synchronizing Merge is associated and it must merge all of the branches emanating from the Multi-Choice. These branches must either flow from the Structured Synchronizing Merge without any splits or joins or they must be structured in form (i.e. balanced splits and joins) [12].

Realization – Structured Synchronizing Merge (WCP-07) can be obtained in NESTFLOW combining a parallel block with a choice block for each branch. As stated for the previous pattern, since NESTFLOW is a block-structured language the implementation of this pattern is the same of Multi Choice (WCP-06). □

WCP-09 Structured Discriminator

Description – The convergence of two or more branches into a single subsequent branch following a corresponding divergence earlier in the process model such that the thread of control is passed to the subsequent branch when the first incoming branch has been enabled. Subsequent enablements of incoming branches do not result in the thread of control being passed on. The Structured Discriminator construct resets when all incoming branches have been enabled. The Structured Discriminator occurs in a structured context, i.e. there must be a single Parallel Split construct earlier in the process model with which the Structured Discriminator is associated and it must merge all of the branches emanating from the Structured Discriminator. These branches must either flow from the Parallel Split to the Structured Discriminator without any splits or joins or they must be structured in form (i.e. balanced splits and joins) [12].

Realization – Structured Discriminator (WCP-09) can be realized in NESTFLOW using a single receive r that waits for a message from several send $s_1, \ldots, s_n$. After completion each task $A_1, \ldots, A_n$ involved in the synchronization sends a message to r which stores the first arrived object in the corresponding variable and continues the execution. Due to the structured context required by the pattern, all involved tasks belong to the same parallel block. For instance, in Fig. 6.19.b tasks A and B belongs to the same parallel block, when at least one of them has completed the process execution can continue with task T, the subsequent termination of the other branch has no effect. The pattern can be executed multiple times preceding it with a reset phase if necessary. □

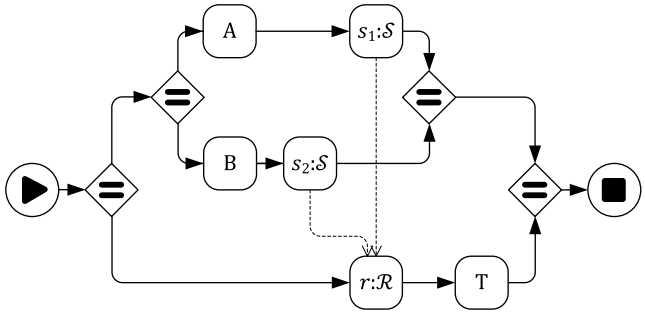


Fig. 6.19. Wcp-09 Structured Discriminator implementation in NestFlow. (a) Simplest case with only two tasks. (b) General case with n tasks.

Wcp-28 Blocking Discriminator

Description – The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. The Blocking Discriminator construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the Blocking Discriminator has reset [12].

Realization – Similarly to Structured Discriminator (Wcp-09), Blocking Discriminator (Wcp-28) can be realized using a single receive r that waits a message from several send $s_1, \ldots s_n$. Each task $A_1, \ldots, A_n$ involved in the synchronization sends after its completion a message to r which stores the first arrived object in the corresponding variable and continues the execution. The only difference with the previous pattern is that the involved tasks can belong to different execution path. For instance, in Fig. 6.20 tasks A and B comes to the same divergency point, while C belongs to another parallel block; when any of this task completes, the execution can continue with T, and the completion of any other of these tasks has no effects. The pattern can be executed multiple times preceding it with a reset phase if necessary. □

Wcp-29 Canceling Discriminator

Description – The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. Triggering the Cancelling Discriminator also cancels the execution of all of the other incoming branches and resets the construct [12].

Realization – Canceling Discriminator (Wcp-29) can be realized in NestFlow using a single receive r that waits for a message from several send comments $s_1, \ldots, s_n$ following the involved tasks and a subsequent send s that cancels the remaining

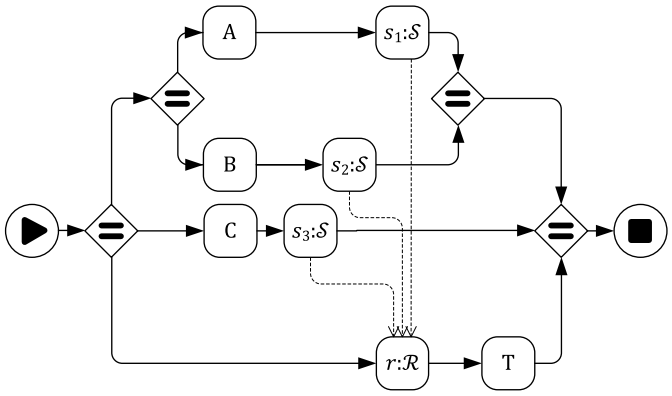


Fig. 6.20. WCP-28 Blocking Discriminator implementation in NESTFLOW.

activities. Each task involved in the synchronization sends after its completion a message to r which stores the first arrived object in the corresponding variable and continues the execution. A cancellation message is sent to all involved task instances and it does not affect the completed instances. For instance, in Fig. 6.21 when any between A and B completes, the send s sends a cancellation message to the involved tasks before continuing the execution with T. The pattern can be executed multiple times preceding it with a reset phase.

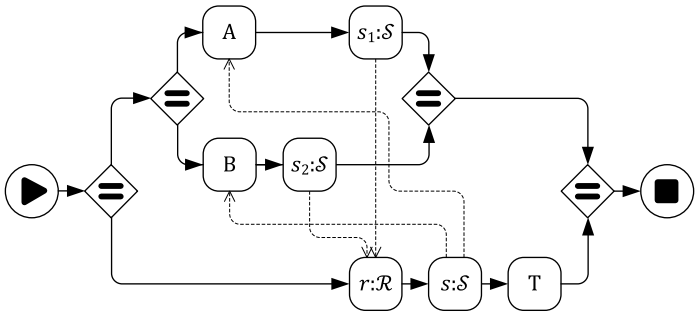


Fig. 6.21. WCP-29 Canceling Discriminator implementation in NESTFLOW.

NESTFLOW offers a more simple yet powerful mechanism for expressing structured canceling discriminator (WCP-X1): the generalized partial join at end of the parallel block allows one to define more sophisticated join conditions on declared variables, not only simple thresholds on the number of joined threads. This feature is not considered here, because WCP-X1 is not part of the original WCPs [12]. □

WCP-30 Structured Partial Join

Description – The convergence of two or more (say m) branches into a single subsequent branch following a corresponding divergence earlier in the process model such that the thread of control is passed to the subsequent branch when k of the

incoming branches have been enabled, where k is less than m. Subsequent enablements of incoming branches do not result in the thread of control being passed on. The join construct resets when all active incoming branches have been enabled. The join occurs in a structured context, i.e. there must be a single Parallel Split construct earlier in the process model with which the join is associated and it must merge all of the branches emanating from the Parallel Split. These branches must either flow from the Parallel Split to the join without any splits or joins or be structured in form (i.e. balanced splits and joins). [12].

Realization – Structured Partial Join (Wcp-30) can be realized similarly to the Structured Discriminator (Wcp-09), simply placing the receive r into a loop. Each task involved in the synchronization sends after its completion a message to r which stores the first arrived object in the corresponding variable and increments the counter i. The receive is performed k times for waiting the completion of exactly k tasks. Due to the structured context required by the pattern, all involved tasks belong to the same parallel block. For instance, in Fig. 6.22 three tasks A, B and C belongs to the same parallel block, after the completion of each of them a message is sent to r which is performed two times in order to wait the completion of exactly two tasks before continuing the execution with T. The subsequent completion of the remaining task has no effect. The pattern can be executed multiple times preceding it with a reset phase if necessary. □

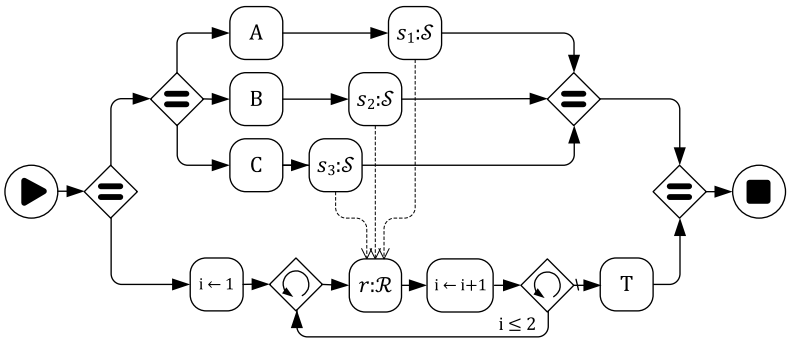


Fig. 6.22. Wcp-30 Structured Partial Join implementation in NestFlow.

Wcp-31 Blocking Partial Join

Description – The convergence of two or more branches (say n) into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when k of the incoming branches has been enabled (where $2 \leq k < m$). The join construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the join has reset. [12].

Realization – Blocking Partial Join (WCP-31) can be realized similarly to the Blocking Discriminator (WCP-28), simply placing the receive r into a loop. After its completion each task involved in the synchronization sends a message to r which stores the first arrived object in the corresponding variable and increments the counter i. The receive is performed k times for waiting the completion of exactly k tasks. As for the Blocking Discriminator, the involved tasks can belong to different execution paths. The example in Fig. 6.23 is very similar to the one in Fig. 6.22 except for the fact that C does not belong to the same parallel block of A and B. When at least two of these three tasks completes, independently from its belonging block, the execution continues with T while the subsequent completion of the remaining task has no effects. The pattern can be executed multiple times preceding it with a reset phase if necessary. □

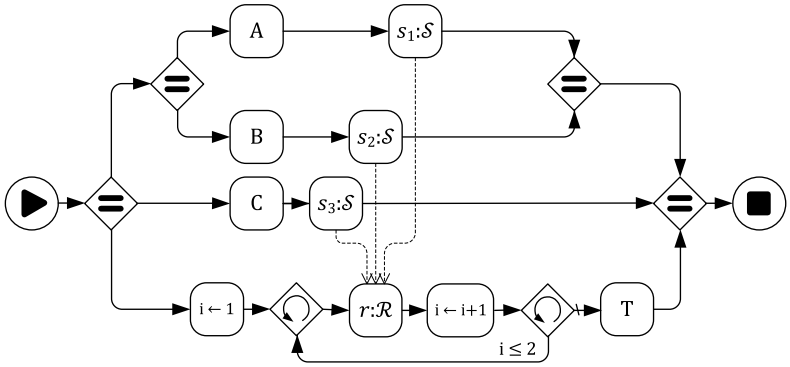


Fig. 6.23. WCP-31 Blocking Partial Join implementation in NESTFLOW.

WCP-32 Canceling Partial Join

Description – The convergence of two or more branches (say n) into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when k of the incoming branches have been enabled where k is less than n. Triggering the join also cancels the execution of all of the other incoming branches and resets the construct. [12].

Realization – Canceling Partial Join (WCP-32) can be realized similarly to the Canceling Discriminator (WCP-29), simply placing the receive r into a loop. Each task involved in the synchronization sends after its completion a message to r which stores the first arrived object in the corresponding variable and continues the execution. A cancellation message is sent to all involved task instances and it does not affect the completed instances. In the example of in Fig. 6.24, the three tasks A, B, and C do not come from a unique divergency point, any time two of them complete and send a message to r, a the send s sends a cancellation message to all branches before continuing the execution with T. The pattern can be executed multiple times preceding it with a reset phase if necessary.

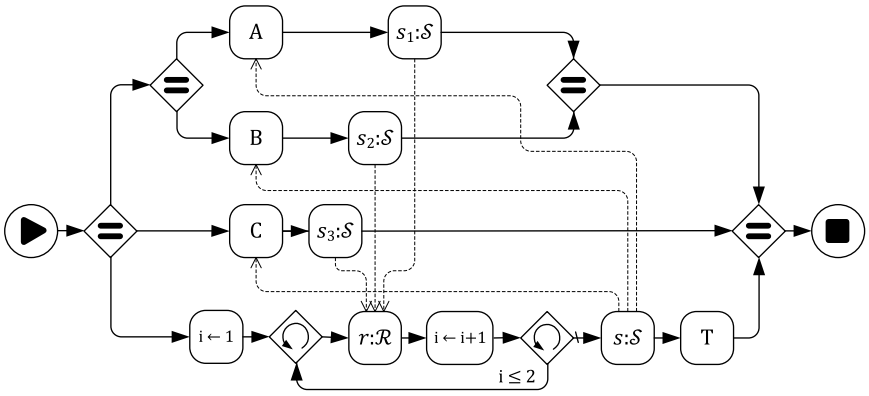


Fig. 6.24. Wcp-32 Canceling Partial Join implementation in NestFlow.

NestFlow offers a more simple yet powerful mechanism for expressing structured canceling partial join (Wcp-X2): the generalized partial join at end of the parallel block allows to define more sophisticated join conditions, not only simple thresholds on the number of joined threads. This feature is not considered here, because Wcp-X2 is not part of the original WCPs [12]. □

Wcp-33 Generalized And-Join

Description – The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled. Additional triggers received on one or more branches between firings of the join persist and are retained for future firings [12].

Realization – Generalized And-Join (Wcp-33) can be obtained in NestFlow using an and-receive r (or a sequence of receive commands) which waits for a message from all the involved tasks. Each task sends after its completion a message to r, r waits for an object from each connected stream before proceeding with the following task T. For instance, in Fig. 6.25 the receive r waits for the completion of A, B and C before executing T independently for the fact that these tasks do not belong to the same parallel block. □

Wcp-37 Acyclic Synchronizing Merge

Description – The convergence of two or more branches which diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled. Determination of how many branches require synchronization is made on the basis on information locally available to the merge construct. This may be communicated directly to the merge by the preceding diverging construct or alternatively it can be determined on the basis of local data such as the threads of control arriving at the merge [12].

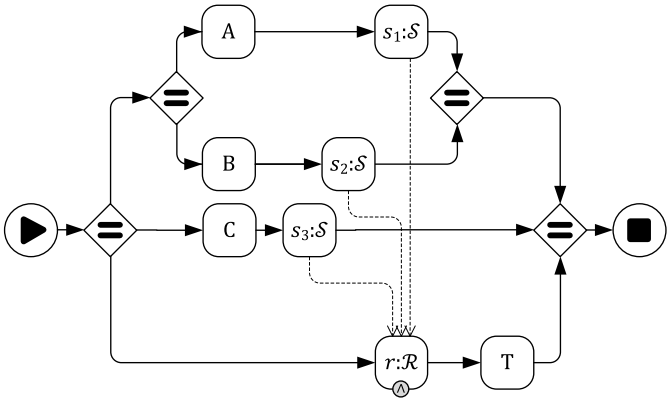


Fig. 6.25. WCP-33 Generalized And-Join implementation in NESTFLOW.

Realization – Acyclic Synchronizing Merge (WCP-37) can be realized by using an and-receive, as for Generalized And-Join (WCP-33), and wrapping each task $A_1, \ldots, A_n$ into a choice block followed by a send. If the choice block condition $\varphi_j(\bar{x}_j)$ evaluates to true, then A_j executes, otherwise a skip is performed; in any case a message is sent to notify the and-receive. In the example of Fig. 6.26, tasks A, B and C does not belong to the same parallel block, anyway task T is performed only after the completion of each of these tasks that are active. □

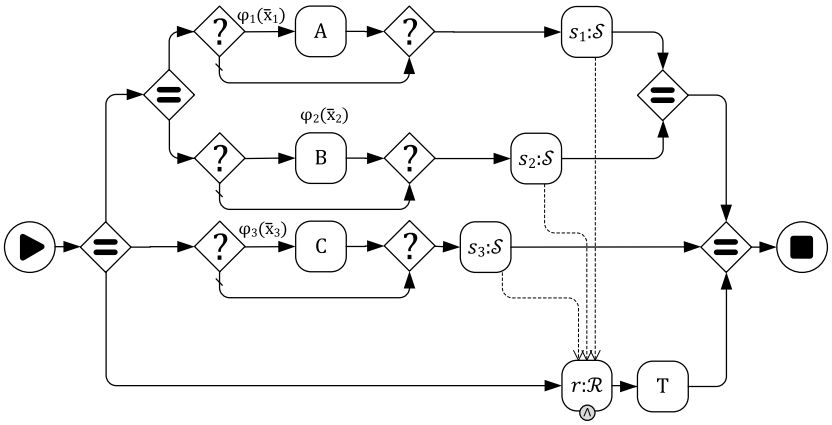


Fig. 6.26. WCP-37 Acyclic Synchronizing Merge implementation in NESTFLOW.

WCP-38 Generalized Synchronizing Merge

Description – The convergence of two or more branches which diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when either (1) each active incoming branch has

been enabled or (2) it is not possible that any branch that has not yet been enabled will be enabled at any future time [12].

Realization – Generalized Synchronizing Merge (Wcp-38) can be realized in NestFlow in the same way of the Acyclic Synchronizing Merge (Wcp-37), because the presence of cycles does not cause any problems. □

Wcp-05 Simple Merge

Description – The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch [12].

Realization – Simple Merge (Wcp-05) can be obtained using a receive r command that waits for an object from one of the connected streams and then execute the following task. Each involved task sends after its completion a message to r which stores the first arrived object in the corresponding variable and immediately executes T. This operation is performed n times, where n is the number of the tasks to be synchronized. The use of a loop block ensures that T is executed multiple times respecting the completion order of the involved task instances but without overlapping executions. Considering the model in Fig. 6.27, when any among A, B or C completes, an execution of T is performed. The task T is performed at most three times in sequence. □

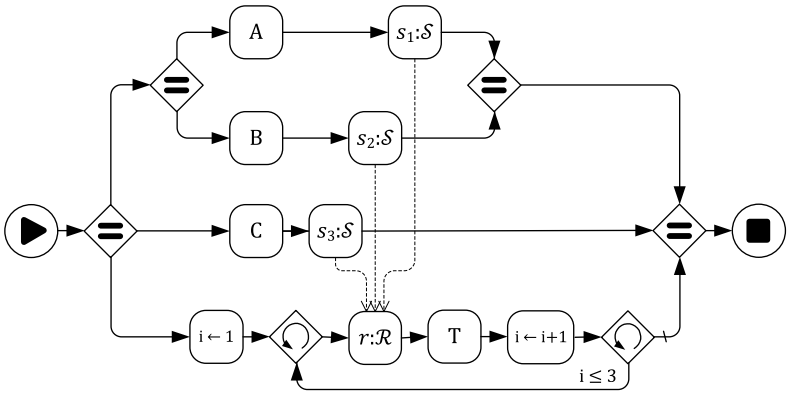


Fig. 6.27. Wcp-05 Simple Merge implementation in NestFlow.

Wcp-08 Multi Merge

Description – The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch [12].

Realization – Multi Merge (WCP-08) can be obtained similarly to Simple Merge (WCP-05) by substituting the task T with a spawn command and wrapping the entire loop block into a concur block, as depicted in Fig. 6.28. Any time one of the involved task instances completes, a message is sent to r and the following spawn command creates a new instance of T, which is immediately executed inside the concur block. This implementation of Multi Merge (WCP-08) is safe: namely, it does not produce overlapping executions of the same instance. □

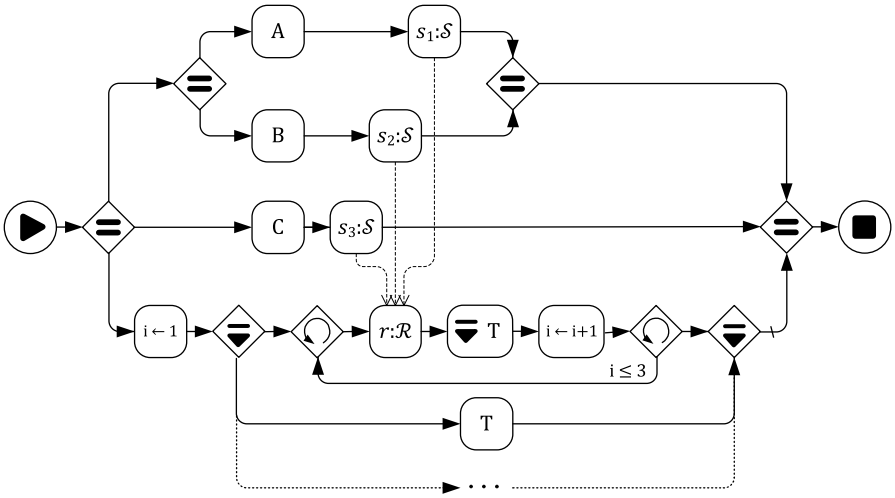


Fig. 6.28. WCP-08 Multi Merge implementation in NESTFLOW.

WCP-41 Thread Merge

Description – At a given point in a process, a nominated number of execution threads in a single branch of the same process instance should be merged together into a single thread of execution [12].

Realization – Thread Merge (WCP-41) is not supported as a matter of principle, because it deals with explicit thread manipulation. □

6.1.8 Evaluation Method

WCPs are a collection of behavioral patterns recurrently found in BP models. They are described in natural language and formalized as Colored Petri Nets (CPNs): a proper extension of Petri Nets enhanced with data, types and functional expressions. A particular WCP can be encoded as a CPN in several different ways; however, if the use of Petri nets constructs is preferred against the use of functional expressions, then the pattern complexity can be roughly quantified by the number of graphical constructs used for representing it. In particular, the WCPs encoding in [12] can be taken as a unit of comparison because it uses basic constructs and it is reasonable to assume that this encoding is nearly optimal, namely no simpler encoding can be given without exploiting advanced features.

WCPs have been used to analyze and classify a great number of BPMLs in order to expose their peculiarities and ease system comparison. In such analysis the commonly accepted scoring assigned to each WCP implementation is *fully supported* (+), if there is a language construct that directly implements the pattern, *partially supported* (±), if there is a construct that produces only a similar behavior to the one prescribed by the pattern, or *unsupported* (−), if none of the provided constructs reproduces the required behavior.

This approach is questionable because it focuses only on language constructs at the expense of the actual behavior that can emerge when such constructs are put together. This leads to paradoxical conclusions: for instance, one should accept that CPNs, the language chosen to formalize all WCPs, supports only few of them; hence, it is not suitable for BP design. The same conclusion holds even more for classical Petri nets that are strictly less expressive than CPNs, despite they are a widely accepted formal language for BP modeling and analysis.

In this paper the NestFlow evaluation is based on how many constructs are needed to reproduce a particular WCP with respect to the CPNs constructs used in [12]. CPNs seem a good reference for comparison because it is hard to conceive a graphical formal language with simpler constructs and equal expressiveness. Any language tailored for BP modeling should generally perform better in encoding WCPs than CPNs, otherwise something goes wrong with the language design.

At first glance, it seems easy to reach the maximum score in a WCPs analysis by conceiving a BPML that provides one construct for each pattern. Although this is ideally possible, one should explain what happens when two or more of these constructs are put together to design a BP model and how they can be used to express emerging patterns not considered in the current WCPs collection.

6.1.9 Evaluation of NestFlow WCPs Support

This section exposes the results of the NestFlow evaluation, giving a first evidence about its expressiveness and suitability for BP design. Multiple-instance patterns are not considered because they can be seen as a specialization of those presented here, where the involved task instances are always of the same type.

The result of this analysis is summarized in Table 6.5: the NestFlow support of each WCP is ranked excellent (★★★), good (★★☆), fair (★☆☆) and none (☆☆☆) using the following parameters: (1) the ratio ρ between the number of

Table 6.5. A summary of NestFlow WCPs support. WCPs are grouped as in [15] and ordered on the basis of their similarities. *Code* is a unique pattern identifier defined in [12] and *Name* is its common name. Given the number of involved tasks n, *NF* is the number of NestFlow constructs used to represent the WCP in the *worst case* scenario, while *CPN* is the number of CPN constructs used in [12]; ρ is the ratio between the values in *NF* and *CPN* columns for large n, and γ is the ratio between the number of used links and the total number of NestFlow constructs. *Eval* is the overall evaluation.

	Code [12]	Name	*NF*	*CPN*	ρ	γ	Eval
G1	Wcp-01	Sequence	$n+2$	$4n+2$	0.25	0.00	★★★
	Wcp-17	Interleaved P.O. Routing	$4n+1^{(a)}$	$14n+6^{(a)}$	0.29	0.25	★★☆
	Wcp-40	Interleaved Routing	$n+4$	$10n+10$	0.10	0.00	★★★
	Wcp-39	Critical Section	$8n+9$	$14n+8$	0.57	0.37	★☆☆
	Wcp-18	Milestone	10	18	0.56	0.10	★★★
G2	Wcp-10	Arbitrary Cycles	—	29	—	—	☆☆☆
	Wcp-21	Structured Loop	4	15	0.27	0.00	★★★
	Wcp-22	Recursion	11	17	0.65	0.18	★★★
G3	Wcp-19	Cancel Activity	14	30	0.47	0.14	★★★
	Wcp-25	Cancel Region	$14n$	—	—	0.14	★★★
	Wcp-20	Cancel Case	14	—	—	0.14	★★★
G4	Wcp-23	Transient Trigger	6	16	0.37	0.17	★★★
	Wcp-24	Persistent Trigger	5	9	0.55	0.20	★★★
G5	Wcp-11	Implicit Termination	0	—	—	—	★★★
	Wcp-43	Explicit Termination	1	—	—	0.00	★★★
G6	Wcp-04	Exclusive Choice	$n+2$	$6n+4$	0.17	0.00	★★★
	Wcp-16	Deferred Choice	$5n+4$	$4n+6^{(b)}$	1.25	0.00	★★★
	Wcp-06	Multi Choice	$4n+1$	$7n+3$	0.57	0.00	★★★
	Wcp-02	Parallel Split	$2n+1$	$7n+3$	0.29	0.00	★★★
	Wcp-42	Thread Split	—	$n+5$	—	—	☆☆☆
G7	Wcp-09	Structured Discriminator	$5n+2$	$5n+14$	1.00	0.20	★★☆
	Wcp-28	Blocking Discriminator	$5n+2$	$9n+17$	0.56	0.20	★★☆
	Wcp-29	Canceling Discriminator	$6n+4$	$10n+15$	0.60	0.33	★☆☆
	Wcp-30	Structured Partial Join	$5n+6$	$5n+14$	1.00	0.20	★★☆
	Wcp-31	Blocking Partial Join	$5n+6$	$9n+17$	0.56	0.20	★★☆
	Wcp-32	Canceling Partial Join	$6n+7$	$10n+15$	0.60	0.33	★☆☆
	Wcp-05	Simple Merge	$5n+8$	$5n+5$	1.00	0.20	★★☆
	Wcp-08	Multi Merge	$7n+9$	$5n+5$	1.40	0.20	★☆☆
	Wcp-07	Structured Synch. Merge	$4n+1$	$9n+6$	0.44	0.00	★★★
	Wcp-37	Acyclic Synch. Merge	$7n+2$	$11n+6$	0.64	0.14	★★☆
	Wcp-38	Generalized Synch. Merge	$7n+2$	—	—	0.14	★★☆
	Wcp-03	Synchronization	$2n+1$	$7n+3$	0.14	0.00	★★★
	Wcp-33	Generalized And-Join	$5n+2$	$7n+3$	0.701	0.20	★★☆
	Wcp-41	Thread Merge	—	$n+5$	—	—	☆☆☆

G1 Sequence Patterns
G2 Repetition Patterns
G3 Cancellation Patterns
G4 Trigger Patterns
G5 Termination Patterns
G6 Branching Patterns
G7 Synchronization Patterns
$^{(a)}$ Assuming P.O. graph size $n-1$
$^{(b)}$ Considering the richer representation

NestFlow constructs used to encode the pattern in the worst case scenario and the number of CPNs constructs used in [12] for representing the same behavior, (2) the ratio γ between the number of links and the number of control-flow constructs used in the NestFlow interpretation and finally (3) the variety of involved NestFlow constructs. The *NF* column contains the total number of used NestFlow constructs, including the number of links and the maximum number of running threads. Similarly, the *CPN* column contains the number of CPNs constructs used in [12], including the number of transitions, places, arcs and the maximum number of involved tokens. For both languages the count does not include additional notations used to specify conditions and expressions. For estimating pattern complexity, a link can be considered a weak control-flow relation manipulated by more reliable control-flow structures; with this interpretation, γ gives a first clue about the level of unstructuredness of the pattern because links may cross the main control-flow structure; in practice, structured forms are always preferred during design, and the number of links will be substantially below γ, which represents a worst case estimation.

Pattern support with worst case $\rho \leq 0.50$ and $\gamma \leq 0.20$ and an optimal use of constructs is ranked excellent (★★★). We also accept in this category pattern implementations with $\rho > 0.50$ or a value of γ near to 0.20 when these ratios do not depend on the number of tasks n. Deferred Choice (Wcp-16) and Multi Choice (Wcp-06) are also ranked excellent even if $\rho > 0.50$, because Wcp-16 has an optimal implementation given by the receive construct and Wcp-06 can be captured by a single specialized construct illustrated in Fig. 5.37 of the previous chapter. Patterns with worst case $\rho \leq 1.00$ and $\gamma \leq 0.20$ and an adequate use of constructs are ranked good (★★☆). Interleaved P.O. Routing (Wcp-17) is also ranked good, because the worst case coefficients $\rho = 0.29$ and $\gamma = 0.25$ are related to an ideal worst case partial order. Patterns with $\rho > 1.00$ or $\gamma > 0.20$ are ranked fair (★☆☆), while the unsupported patterns are ranked none (☆☆☆). Such criteria allows a more fine-grained and measurable evaluation of WCP support. Nevertheless, a pattern-based analysis remains a *qualitative* evaluation of expressiveness: a WCP specifies a system behavior that can be obtained in different ways, combining more or less sophisticated constructs. For instance, in modeling a process by CPNs we can obtain a particular behavior mainly using Petri nets constructs or alternatively using few graphical constructs, encoding most of the logic in functional arc expressions and transition guards. The considered language constructs are also important but a pattern-based evaluation that puts too much emphasis on constructs, instead of on the overall behavior, leads to paradoxical conclusions as explained in the previous section. The NestFlow evaluation is mainly based on the effort needed to replicate WCPs behavior: for each pattern such effort has been quantified by comparing the NestFlow implementation in the worst case scenario with the CPNs reference implementation [12]. The count of CPNs constructs is omitted for those patterns whose CPNs model is an ad-hoc construction that cannot be easily quantified, for instance because the number of constructs depends on the reachable states.

Any high-level BPML that wants to support WCPs will likely provide a more compact representation of these patterns, with more specific constructs than CPNs. Except for few patterns, the NestFlow ratio ρ is always less than one;

more specifically, it usually needs half of the CPNs constructs for expressing the same behavior. In some cases the ratio ρ is greater than one or the pattern is not supported at all, as for Multi Merge (WCP-08), Arbitrary Cycles (WCP-10), Thread Merge (WCP-41) and Thread Split (WCP-42). The lack of support for these patterns is acceptable because consistent with the initial design intentions: NESTFLOW is built to be modular and these patterns hinder modularity.

◊

6.2 Geo-Processing Application

The term *geo-processing* denotes long-running interactive computations that rely on self-contained, specialized and interoperable services.

In [22] the authors evaluate the applicability of existing WfMSs for supporting such kind of processes. In particular, they consider two kinds of systems classified as business WfMSs and scientific WfMSs, taking YAWL and Kepler as representative. They state that processes in the geographical field can benefit from both the adoption of business and scientific WfMSs, but none of them is completely satisfactory. The found limitations regard three different aspects: modeling, visualization, and processing of spatial data. The latter is the most serious one, because it cannot be solved by simply adding features to existing WfMSs.

In this kind of process, two forms of parallelism are distinguished: functional and data decomposition. Functional decomposition techniques deal with the distribution of operations among the available resources. It can be achieved by decomposing complex computation activities into smaller parts (tasks), defining the execution order of these operations and the dependencies between them, so that some operations can be performed in parallel while others in sequence. On the other hand, data decomposition techniques subdivide data into independent chunks: in this way, multiple instances of the same activity can be executed in parallel on different inputs. The partial results produced by each activity instance are finally combined to form the overall output. PAISs are more suitable for representing the functional decomposition of processes and the interactions among different agents. At any step the workflow engine can monitor the overall state of the process, which activities have been performed and which are in execution. On the other hand, this kind of systems provides a poor support for the data decomposition, which is essential for intensive computations. Optimizations at such level have to be addressed in an ad-hoc manner by the underlying software layer. Conversely, scientific WfMSs have been developed for supporting intensive computations and can easily exploit the use of Grid technologies in a transparent way for the user. Scientific WfNSs have been studied to simplify the composition of computational blocks and, hence, they offer a better chance to provide a library for geo-processing. However, some problematic aspects remain, for instance how to provide to the user a complete overview of the process execution, or how to seamlessly integrate cross-cutting concerns like resource management.

In other words, the interactive nature of long-running geo-processing activities, the importance of domain expert knowledge in driving the computation and the need to coordinate the effort of different agents, are better addressed by business WfMSs. On the contrary, scientific WfMSs can provide a support for intensive long-running computations required by geo-processes.

The author concludes that an ideal WfMS for geo-processing has to combine the characteristics of both approaches in a coherent system. In particular, the data-flow computation model adopted by scientific WfMSs enhanced with coarse-grained control-flow constructs, in order to express the control-flow logics that emerges from fine-grained data-flow relations. This can be useful for mitigating the main drawback of using a data driven approach with respect to the control-

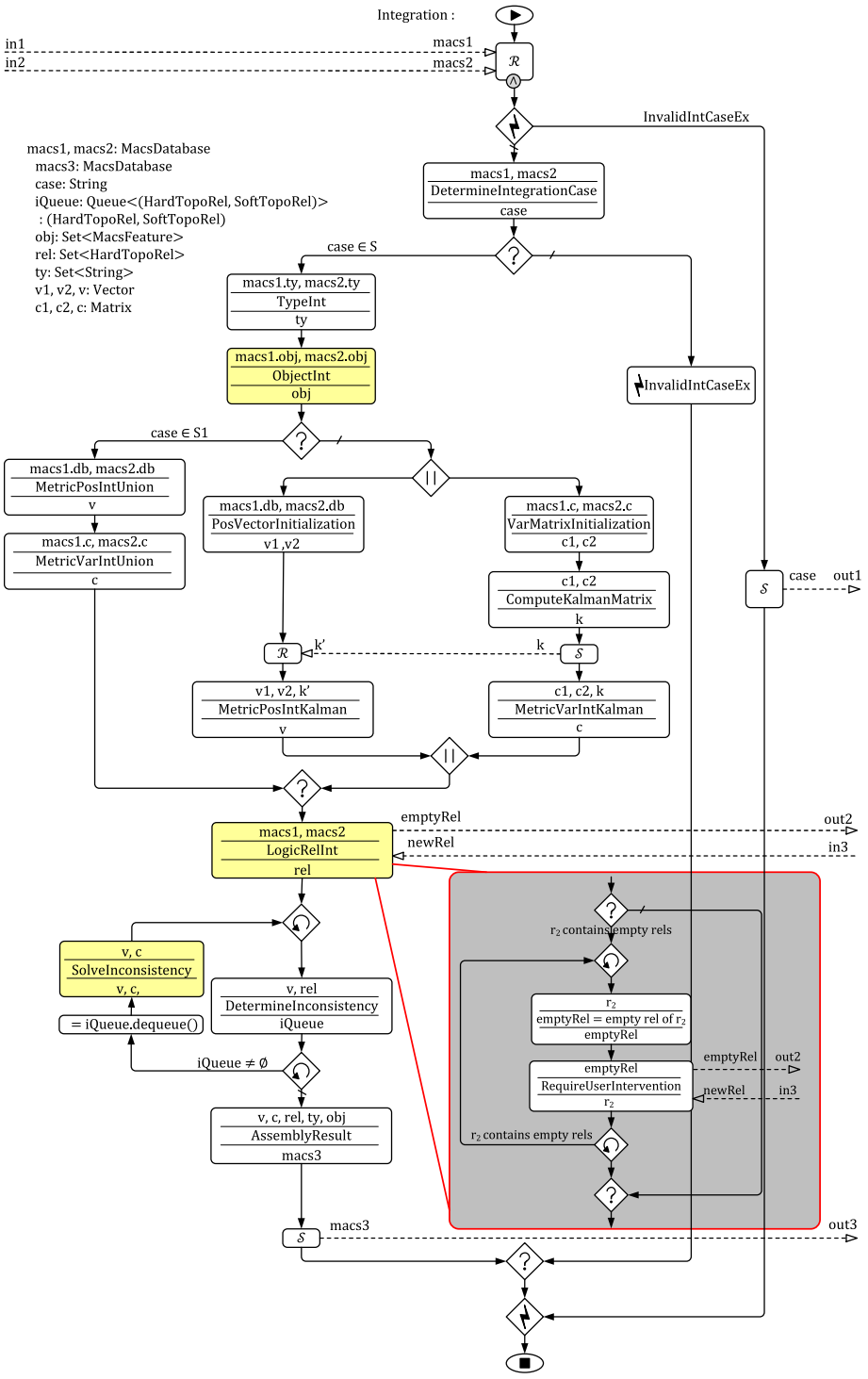


Fig. 6.29. Example of geographical process.

flow one offered by PAISs, namely the loss of easy to grasp information about the overall process execution.

For all these reasons in [118] the suitability of NestFlow for geo-processing has been evaluated by considering the modeling of a representative process. In this work NestFlow data types have been extended to represent basic geometric primitives and a specialized library of functionalities have been implemented. Fig. 6.29 depicts an example of geographical process presented in [118], which performs the integration between two distinct datasets. The stream and variable declarations on the left exemplify the use of some specific geographical types, such as features and topological relations, while composite tasks are highlighted in yellow.

The process starts when two messages are received, containing the reference to the two datasets to be integrated: for this reason the first receive is decorated with a logic *and* ($\wedge$) symbol. The subsequent try block is used to manage the case in which an improper input has been provided: in this case an exception is thrown and the corresponding try branch is executed, which notifies the external environment through a send.

Subsequently, a parallel block is used to specify activities that can be performed in parallel: anyway, such activities have to synchronize at some point, because one branch needs data produced by the other one. This synchronization is represented by a message exchange, making explicit the nature and reason of the interaction, using another PML, such as YAWL or BPMN, such interaction can be represented only using an unstructured construction.

Another important aspect is related to the decomposition of task `LogicRelInt` which has an input and an output stream connected to it, while its internal structure is reported inside the gray box. Notice that the data provided through the input stream are not required at the beginning and in certain cases the process can complete without requiring it. Similarly, data delivered through the output stream are produced during the computation, not at completion, and in some cases the computation can terminate without producing any message. This decomposition cannot be easily obtained using a classical PML, such as YAWL, because it requires that input data are provided at the beginning, and output data are produced only at completion.

$\diamond$

6.3 Health-Care Application and Controllability

WfMSs have been increasingly applied for designing and executing medical processes [119]. This kind of processes presents some distinctive characteristics which regards the need to model data dependencies among tasks and the need to represent temporal constraints on task executions. For instance, relatively to data aspects, a surgery intervention could need the results of the concurrent bioptic analysis to be properly concluded, and this data synchronization needs to be explicitly represented. Similarly, as regards to temporal aspects, temporal constraints are always present and their management is mandatory to successfully complete a medical process. For example, to successfully apply the fibrinolytic therapy to patients with ST-segment Elevation Myocardial Infarction (STEMI), a maximum delay must be respected from the admission to the emergency department. Finally, since the description dimension of medical processes is often huge, it is important to prevent the possibility of having deadlocks and/or lacks of synchronization; hence, the use of structured PMLs is recommended in place of unstructured ones.

For all these reasons, in [23] the authors study an extension of NestFlow, called Tnest, that allows one to represent both data aspects and temporal constraints using a structured PML, whose expressiveness is not reduced w.r.t. the widely used unstructured languages. In particular, two main kinds of temporal constraints can be defined in Tnest: activity duration and relative constraint.

In order to define both kinds of temporal constraints the concept of edge, connector, and task are introduced. An *edge* represents a control-flow relation between two (terminal or non-terminal) blocks. A *connector* denotes either the entry or the exit construct of a non-terminal block. A *task* is essentially a more or less complex pre-existing process specification invoked using run command or created at run-time with a spawn command.

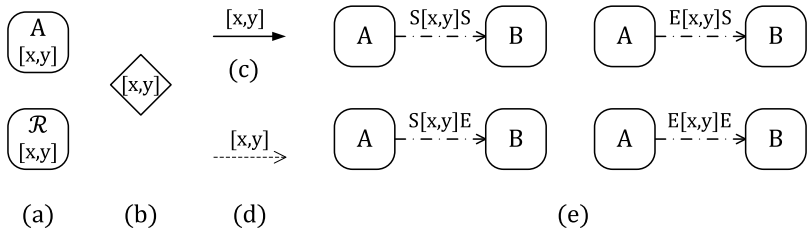


Fig. 6.30. NestFlow constructs with temporal constraints. (a) Duration constraint associated to a task, or a `receive` command, respectively. (b) Duration constraint associated to a connector, notice that the connector is empty to be general. (c) Delay defined on a control-flow relation. (d) Delay defined on a link. (e) Relative constraint between two tasks.

An *activity duration* represents the allowed temporal spans for the execution of either a command, a connector, an edge, or a link as depicted in Fig. 6.30.a, Fig. 6.30.b, Fig. 6.30.c, and Fig. 6.30.d respectively. Except for spawn, send, and throw, each command is assumed to have a duration specified by the designer using a notation like `[MinD,MaxD] Granularity`, where $0 \leq$ `MinD` $\leq$ `MaxD` $\leq \infty$

and `Granularity` stands for the used time unit. If the designer does not specify an activity duration, it is assumed to be $[1, \infty]$ `MinGranularity`, where `MinGranularity` is the minimum available time granularity. Conversely, `spawn`, `send`, and `throw` are non-blocking activities and are used as milestones to start other blocks or messages; hence, they are assumed to have fixed duration that cannot be modified by the designer. Moreover, the model allows also the setting of a duration for edges and links. The duration associated to an edge can be viewed as a *delay*, because it represents the allowed delay to enact the execution of the second block after the end of the execution of the first one. Similarly, the duration associated to a link represents the allowed *delivery time* of a message once it is generated. It is assumed that the duration constraint associated to a task is not modifiable, since tasks are executed by external agents usually requiring a non negotiable time to be completed, while the duration associated to a connector, edge and link can be modified in order to satisfy all the given requirements.

The other kind of temporal constraints are the *relative constraints*. They allow the expression of several temporal constraints among activities. Differently from the duration constraints, relative constraints are not mandatory. A relative constraint limits the time distance between the starting/ending instants of two non-consecutive blocks. It is graphically represented as a dash-dot-dash edge between two blocks, as reported in Fig. 6.30.e. The label on the edge specifies the constraints according to the following pattern: $\langle I_F \rangle$`[MinD,MaxD]`$\langle I_S \rangle$ `Granularity`, where $\langle I_F \rangle$ marks the instant of the *first* activity to use, and it can be equal to `S` or `E` corresponding to the start or the end execution instant of the activity, respectively; $\langle I_S \rangle$ marks the instant related to the *second* activity in the same way, and `[MinD,MaxD]` `Granularity` represents the allowed range for the time distance between the two instants $\langle I_F \rangle$ and $\langle I_S \rangle$. It is assumed that $-\infty \leq$ `MinD` $\leq$ `MaxD` $\leq \infty$; in particular, a finite positive `MaxD` value models a *deadline*, since it corresponds to the maximum global allowable execution time for the activities that are present on possible flows between the first node and the second one. On the other hand, a finite positive `MinD` represents the minimum execution time that has to be spent before proceeding after $\langle I_S \rangle$: if the global time spent to execute all activities between $\langle I_F \rangle$ and $\langle I_S \rangle$ is less than `MinD`, then the WfMS has to dynamically manage a suitable action (e.g. to sleep) that depends from the specific applications. A finite negative `MaxD` value expresses the fact that that the $\langle I_S \rangle$ has to occur at least |`MaxD`| instants before $\langle I_F \rangle$. In general, if a designer does not specify the granularity of a range, it is assumed that the granularity is `MinGranularity`.

Since the relative constraint concept is quite general and some blocks have a complex behavior, some unfitting settings are possible. Hence, the following five general construction rules have been defined:

1. Relative constraints cannot be set among activities belonging to mutually exclusive flows.
2. An implicit `E`$[0, \infty]$`S` relative constraint between a `spawn` command and the dotted task generated by it is always set as shown in Fig. 6.31.a. Further constraints between other tasks and the spawned one may be arbitrarily set.
3. Inside a loop, it is possible to set a relative constraint only between tasks but on different cycles of the loop using a specific notation and with some limitations. For example, in Fig. 6.31.a the relative constraint between B and

D has label $\blacktriangleright\mathtt{S}[x,y]\mathtt{S}$ meaning that any instance of D has to start in $[x,y]$ time units after the start of any block B spawned before it.

4. Between two connected send-receive commands there is always an implicit $\mathtt{S}[u,\infty]\mathtt{E}$ constraint where u could be 0 or the lower bound of the duration constraint of the link, in case it is specified. Such constraint has a different meaning w.r.t. the duration constraint of the link: even if the two constraints are graphically similar, while the duration represents the allowed delivery time, the relative constraint dictates the time range limits in which the message has to be produced and consumed, i.e., the *validity time* of a message. A designer can always customize the validity constraint observing that the validity lower bound has to be always not less than the lower bound of the corresponding delivery time.
5. Inside a try block, an implicit $\mathtt{E}[0,\infty]\mathtt{S}$ relative constraint between the throw command and the first task on the associated exception branch it is always set as shown in Fig. 6.31.c. Relative constraints between tasks on an exception branch and tasks outside the try block are not allowed, because it is assumed that the exceptions should hardly occur and allowing relative constraints between tasks on an exception branch and outside tasks makes the temporal checks harder for most of the model executions uselessly.

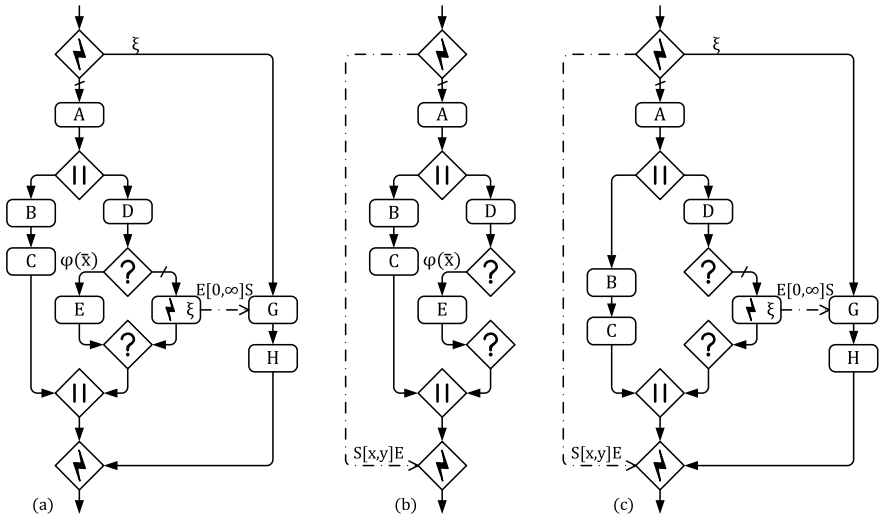


Fig. 6.31. (a) Specification of temporal constraints in presence of `spawn` command. (b) Specification of temporal constraints between a `send` and a `receive`. (c) Specification of temporal constraints in presence of a `catch` block.

An algorithm has also be defined for checking the controllability of a TNEST process. More specifically, *controllability* is the capability of executing a workflow for all possible durations of all tasks and satisfying all temporal constraints. A future work regards the exploiting of NESTFLOW modularity in the specification of temporal constraints, in order to reduce the algorithm complexity.

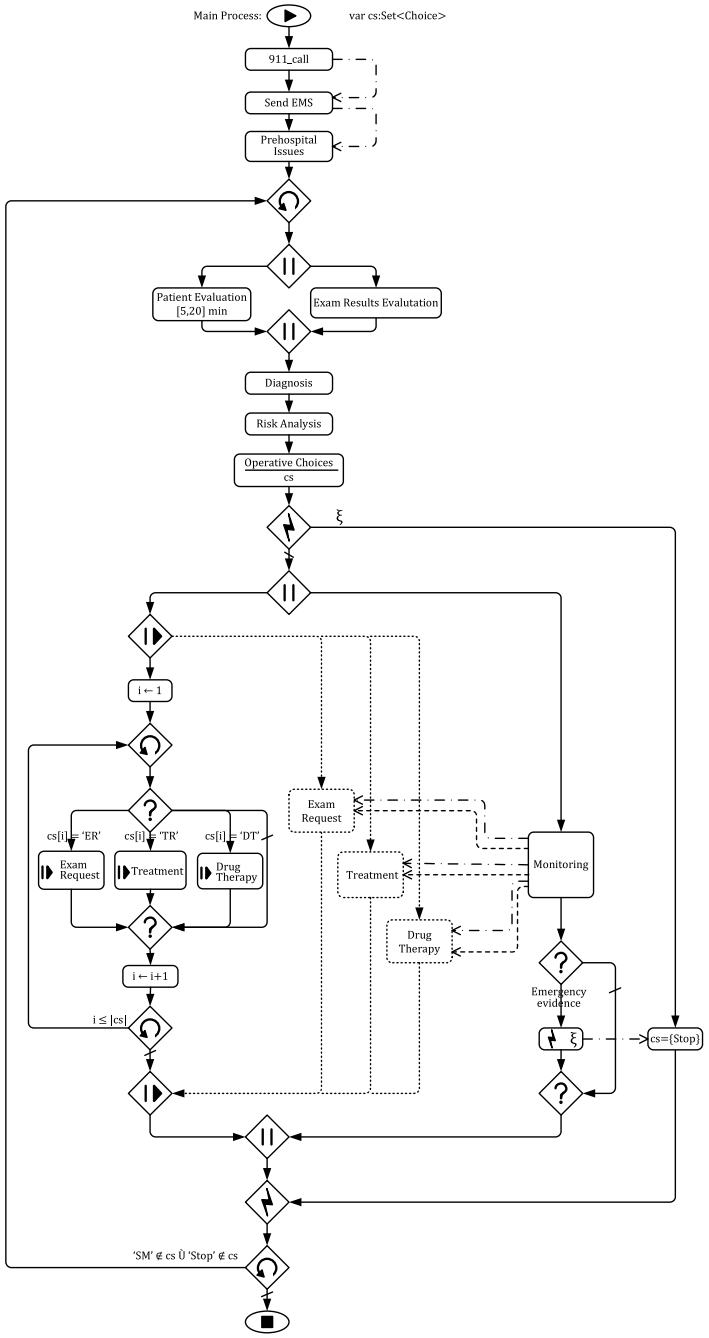


Fig. 6.32. Main process for the diagnosis and treatment of STEMI. 'SM' stands for "Secondary Management", 'ER' stands for "Exam Request", 'DR' stands for "Drug Therapy".

The schema in Fig. 6.32 is an example of application of TNEST for the modeling of heath-care processes involving temporal constraints. In particular, it summarizes the main steps for the diagnosis and treatment of a myocardial infarction (STEMI) described in [120]. Some temporal constraints are defined on the preliminary activities: the EMS (Emergency Medical Services) has to be sent within 2 minutes from the initial 911 call, while pre-hospital treatments have to starts 8 minutes from the EMS arrival. After an initial evaluation of the patient conditions and the available exam results, a first diagnosis is formulated and some operative choices are taken by clinicians. These operative choices may regard the request for other analysis, the execution of some treatments and the administration of suitable drugs. Several instances of such activities can be activated in parallel through the spawn commands contained in the inner loop block. Task Monitoring deals with all activities performed in a stable way for continuously checking the patient conditions during her hospital stay. This task periodically sends information about the patient vital signs to the operative activities that are executing. Temporal constraints are defined on messages and they specify the patient vital signs have to be communicated within 10 minutes and they have a validity of 1 day. In case a complication happens, an exception is thrown and all the running activities are interrupted, in order to immediately start a new complete patient evaluation and define a new diagnosis. The main process terminates when a "secondary management" operative choice is taken.

◊

6.4 Summary and Concluding Remarks

Well established business process modeling practices and empirical research experiments suggested that structured control-flow forms are always desirable to enhance comprehensibility and modularity, and to reduce the probability of introducing subtle errors in process models that are hard to spot and fix without refactoring the entire model. Unfortunately, existing PMLs are not able to support a fully structured control-flow design without losing expressiveness.

WCPs are a well accepted framework for evaluating the expressiveness and suitability of PMLs in the PAIS domain; therefore, they are used at the beginning this chapter for evaluating the suitability and expressiveness of NestFlow. Anyway, the scoring method adopted in a WCP-based analysis is biased towards language constructs, limiting its applicability; hence, in this chapter a more objective evaluation method is introduced: WCP support is evaluated on the basis of the number of used constructs with respect to the CPNs reference implementation proposed in [12], the number of links with respect to the total number of used constructs and the constructs variety. The analysis is performed, whenever possible, for a large number of involved tasks in the worst case scenario. The CPNs language has been chosen for the comparison because it is the language used to formally specify the WCP behaviour. In general, NestFlow supports the majority of WCPs with less constructs than CPNs; it provides a poor support only for those patterns regarding the explicit manipulation of threads, but this is intentionally because NestFlow avoids the explicit thread manipulation as a matter of principle.

The last part of the chapter introduces two other application domains for NestFlow: the geographical and the health-care one. Regarding the design of geographical processes, the main exploited characteristic of NestFlow is the ability to mix a data-flow oriented approach with a control-flow one. Conversely, in the health-care domain, an extension called Tnest is introduced, which allows one to represent various temporal aspects and verify the temporal controllability of an entire process.

7

Conclusion

This thesis begins by giving in Chap. 3 an overview about the state of the art of graphical PMLs. In particular, it firstly consider the Petri nets formalism which is the simplest modeling language able to represent many interesting aspects about processes, and constitutes the theoretical foundation of many PMLs. Petri nets are examined starting from its most generic variant, called here PTNs, from which the other Petri nets languages used through the entire thesis can be obtained as a syntactical restriction. Other three important languages are then analyzed by discussing their essential features: CPNs, YAWL and BPMN. A future work in this direction regards a similar formalization of scientific WfMSs, recalling the results contained in the report [35].

The thesis then focuses in Chap. 4 on verification and correction of process models. The chapter initially discusses some issues that affect the currently available verification techniques and proposes some enhancements to the original notion of soundness in order to easy the correction activity. The focus of the chapter is to propose a novel technique able to support the end-user during the following correction phase. In particular, given a WFNs model with some errors and the information collected during the soundness check, the proposed technique, called PNSA, tries to find a set of useful hints to rectified the original model. Some preliminary experimental results have been presented about the application of such technique to a set of real processes, as a future work the technique will be submitted to an end-user validation in order to verify that the proposed solutions not only contain less errors of the original one, but are considered relevant for the designer. The technique is fairly general and can be easily extended to other modeling languages, such as YAWL or BPMN, eventually by mapping them to WFNs, or to other kinds of soundness definition.

The thesis takes then a different path in Chap. 5 by proposing an alternative modeling approach which is based on the adoption of a structured PML which can guarantee the presence of interesting control-flow properties by construction, instead of using sophisticated verification methods a posteriori. In particular, the NestFlow modeling language is introduced as a proof-of-concept with the aim to prove that structured a PML can be effectively built taking modularity and comprehensibility as primary concerns. Such language is based on a set of new constructs that combine block-structured control-flow constructs with AMP ab-

stractions. Several missing aspects can be added to the language for obtaining a fully-fledged system; for instance, the possibility of sending process instances as objects in streams to support mobility. Other aspects that can be considered in the future regards the flexibility of process models [4], i.e. the possibility of changing processes already in execution, the management of compensation activities.

The thesis concluded by discussing in Chap. 6 the expressiveness and applicability of NestFlow both in the PAIS and other domains. Relatively to the PAIS domain, its expressiveness is evaluated in terms of Worfklow Control-Flow patterns (WCPs) since they have been widely adopted in literature for evaluating and comparing workflow systems. Other patterns have been proposed in literature for evaluating a workflow system, they can be considered in the future for performing additional evaluations, especially when the cited missing features will be added to the language.

The NestFlow application to the geographical domain seems to be promising. A future work in this direction consists in providing some constructs for manipulating the specific domain data in a more effective way, for instance by using spatial query predicates in conditions.

The Tnest extension allows one to specify several interesting temporal constraints on the model execution and check their controllability. A future work in this direction consist in exploiting NestFlow modularity for reducing the complexity of the verification algorithm by providing approximated results.

References

1. Edward A. Lee. Computing Needs Time. *Communication of the ACM*, 52(5):70–79, 2009.
2. Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., 1st edition, 1996.
3. D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Classics in Software Engineering*, pages 139–150, 1979.
4. M. Reichert and B. Weber. *Enhancing Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies*. Springer-Verlag, 2012.
5. Edward A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, 2006.
6. M. Dumas, W. M. P. van der Aalst, and A. H. M. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software Through Process Technology*. Wiley-Interscience, 2005.
7. Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag New York, Inc., November 2007.
8. Wil van der Aalst and Kees van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2004.
9. David Gelernter and Nicholas Carriero. Coordination Languages and their Significance. *Communication of the ACM*, 35(2):96–107, 1992.
10. David Cohn and Richard Hull. Business Artifacts: A Data-centric Approach to Modeling Business Operations and Processes. *IEEE Data Engineering Bulletin*, 32(3):3–9, 2009.
11. Wil M. P. van der Aalst, Mathias Weske, and Dolf Grünbauer. Case Handling: A New Paradigm for Business Process Support. *Data Knowledge and Engineering*, 53(2):129–162, 2005.
12. N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns: a Revised View. Technical Report BPM-06-22, BPM Center Report, 2006.
13. Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.
14. T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580.
15. Arthur H.M. ter Hofstede, Wil M.P. van der Aalst, Michel Adams, and Nick Russell. *Modern Business Process Automation: YAWL and its Support Environment*. Springer-Verlag, November 2009.
16. Object Management Group (OMG). *Business Process Modeling Notation (BPMN) 2.0 (Beta 1)*, August 2009. http://www.omg.org/spec/BPMN/2.0/.

17. B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler. On Structured Workflow Modelling. In *12th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 431–445, 2000.
18. M. Gambini, M. La Rosa, S. Migliorini, and A.H. M. Ter Hofstede. Automated Error Correction of Business Process Models. In *Proceedings of the 9th International Conference on Business Process Management (BPM'11)*, pages 148–165, 2011.
19. C. Combi and M. Gambini. Flaws in the Flow: the Weakness of Unstructured Business Process Modeling Languages Dealing with Data. In *17th International Conference on Cooperative Information Systems (CoopIS'09)*, pages 42–59, 2009.
20. C. Combi, M. Gambini, and S. Migliorini. Towards Structured Business Process Modeling Languages. In *Proceedings of 15th International Conference Advances in Databases and Information Systems (ADBIS'11)*, pages 1–10, 2011.
21. C. Combi, M. Gambini, and S. Migliorini. The NestFlow Interpretation of Workflow Control-Flow Patterns. In *Proceedings of 15th International Conference on Advances in Databases and Information Systems (ADBIS'11)*, pages 316–332, 2011.
22. S. Migliorini, M. Gambini, A. Belussi, M. Negri, and G. Pelagatti. Workflow Technology for Geo-Processing: the Missing Link. In *Proceedings of the 2nd International Conference and Exhibition on Computing for Geospatial Research & Application, COM.Geo 2011*, page 36, 2011.
23. C. Combi, M. Gambini, S. Migliorini, and R. Posenato. Modelling temporal, data-centric medical processes. In *ACM International Health Informatics Symposium (IHI'12)*, pages 141–150, 2012.
24. Elliott Mendelson. *Introduction to Mathematical Logic*. Chapman & Hall/CRC, 5th edition, 2009.
25. Volker Diekert. *The Book of Traces*. World Scientific Publishing Co., Inc., 1995.
26. Oracle GlassFish, Open-Source Application Server. http://glassfish.java.net. Accessed June 2012.
27. Redhat JBoss jBPM, JBoss Open-Source Application Server. http://www.jboss.org/jbpm/. Accessed June 2012.
28. Workflow Patterns Initiative. http://workflowpatterns.com. Accessed June 2012.
29. N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical Report BPM-06-22, BPM Center Report, 2006. http://bpmcenter.org/wp-content/uploads/reports/2007/BPM-07-05.pdf. Accessed June 2012.
30. N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Workflow Data Patterns: Identification, Representation and Tool Support. In *Proceedings of the 24th International Conference on Conceptual Modeling (ER 2005)*, pages 353–368, 2005.
31. N. Russell, W. M. P. van der Aalst, A. H. M. ter Hofstede, and D. Edmond. Workflow Resource Patterns: Identification, Representation and Tool Support. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE'05)*, 2005.
32. P. Wohed, W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, and N. Russell. On the Suitability of BPMN for Business Process Modelling. In *Proceedings of the 4th International Conference Business Process Management (BPM 2006)*, pages 161–176, 2006.
33. P. Wohed, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. Analysis of Web Services Composition Languages: The Case of BPEL4WS. In *Proceedings of the 22nd International Conference on Conceptual Modeling (ER 2003)*, pages 200–215, 2003.
34. N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P van der Aalst. newYAWL: Achieving Comprehensive Patterns Support in Workflow for the Control-flow, Data

and Resource Perspectives. Technical Report BPM-07-05, BPM Center, 2007. http://www.bpmcenter.org Accessed March 2012.
35. S. Migliorini, M. Gambini, M. La Rosa, and A.H.M. ter Hofstede. Pattern-Based Evaluation of Scientific Workflow Management Systems. Technical Report 39935, Queensland University of Technology, 2011.
36. Egon Börger. Approaches to modeling business processes: a critical analysis of BPMN, workflow patterns and YAWL. *Software and System Modeling*, 11(3):305–318, 2012.
37. B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computatition: Practice and Experience*, 18(10):1039–1065, 2006.
38. T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. Pocock, A. Wipat, and P. Li. Taverna: a Tool for the Composition and Enactment of Bioinformatics Workflows. *Bioinformatics*, 20:3045–3054, 2004.
39. S. Majithia, M. Shields, I. Taylor, and I. Wang. Triana: A Graphical Web Service Composition and Execution Toolkit. pages 514–521, 2004.
40. Paul Grefen, Barbara Pernici, and Gabriel Sanchez, editors. *Database Support for Workflow Management: The WIDE Project*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.
41. F. Casati, P. Grefen, B. Pernici, G. Pozzi, G. Snchez, and B. Pernici. WIDE Workflow Model and Architecture. Technical Report CTIT 96-19, University of Twente, 1996. http://www.wi.uni-muenster.de/improot/imperia/md/content/wi-information_systems/lehrveranstaltungen/lehrveranstaltungen/bpmundwfm/ws0405/casati96wide.pdf. Accessed June 2012.
42. Guy Redding, Marlon Dumas, Arthur H. M. ter Hofstede, and Adrian Iordachescu. Transforming Object-Oriented Models to Process-Oriented Models. In *Business Process Management Workshops (BPM 2007)*, pages 132–143, 2007.
43. Dominic Müller, Manfred Reichert, and Joachim Herbst. Data-Driven Modeling and Coordination of Large Process Structures. In *On the Move to Meaningful Internet Systems (OTM 2007)*, pages 131–149, 2007.
44. Dov Dori. Object-Process Methodology. In *Encyclopedia of Knowledge Management*, pages 1208–1220. IGI Global, 2011.
45. Vera Künzle and Manfred Reichert. PHILharmonicFlows: towards a framework for object-aware process management. *Journal of Software Maintenance*, 23(4):205–244, 2011.
46. Manfred Reichert and Peter Dadam. $\text{ADEPT}_{\text{flex}}$ – Supporting Dynamic Changes of Workflows Without Losing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.
47. Peter Dadam and Manfred Reichert. The ADEPT project: a decade of research and development for robust and flexible process support. *Computer Science – Research and Development*, 23(2):81–97, 2009.
48. Jörg Desel and Gabriel Juhás. "What Is a Petri Net?". In *Unifying Petri Nets, Advances in Petri Nets*, pages 1–25, London, UK, UK, 2001. Springer-Verlag.
49. Carl Adam Petri. *Communication with automata*. PhD thesis, Universitt Hamburg, 1966.
50. Claude Girault and Rudiger Valk. *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*. Springer-Verlag New York, Inc., 2001.
51. Gabriel Juh´as, Fedor Lehocki, and Robert Lorenz. Semantics of petri nets: a comparison. In *Winter Simulation Conference (WSC'07)*, pages 617–628, 2007.
52. Robert Jan van Glabbeek. The Individual and Collective Token Interpretations of Petri Nets. In *Concurrency Theory (CONCUR'05)*, pages 323–337, 2005.

53. Eric Badouel, Marek A. Bednarczyk, and Philippe Darondeau. Generalized Automata and Their Net Representations. In *Unifying Petri Nets, Advances in Petri Nets*, pages 304–345, 2001.
54. Julia Padberg and Hartmut Ehrig. Parameterized Net Classes: A Uniform Approach to Petri Net Classes. In *Unifying Petri Nets, Advances in Petri Nets*, pages 173–229, 2001.
55. Rüdiger Valk. Self-Modifying Nets, a Natural Extension of Petri Nets. In *Proceedings of the 5th Colloquium on Automata, Languages and Programming*, pages 464–476, 1978.
56. Rüdiger Valk. On the Computational Power of Extended Petri Nets. In *7th Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 526–535, 1978.
57. Catherine Dufourd, Alain Finkel, and Ph. Schnoebelen. Reset Nets Between Decidability and Undecidability. In *25th International Colloquium on Automata, Languages and Programming*, pages 103–115, 1998.
58. Luca Bernardinello and Fiorella de Cindio. A Survey of Basic Net Models and Modular net Classes. In *Advances in Petri Nets 1992*, pages 304–351, 1992.
59. Robin Milner. *Communicating and Mobile Systems - the Pi-Calculus*. Cambridge University Press, 1999.
60. Toshiro Araki and Tadao Kasami. Some decision problems related to the reachability problem for petri nets. *Theoretical Computer Science*, 3(1):85 – 104, 1976.
61. Michael Hack. Petri Net Languages. *Computation Structures Group - Memo No. 124*, June 1975.
62. Gianfranco Ciardo. Petri Nets with Marking-Dependent Arc Cardinality: Properties and Analysis. In *Advances in Petri Nets 1994*, pages 179–198, 1994.
63. Wolfgang Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., 1985.
64. Catherine Dufourd, Petr Jancar, and Ph. Schnoebelen. Boundedness of reset p/t nets. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming*, ICAL '99, pages 301–310, London, UK, UK, 1999. Springer-Verlag.
65. Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML - Revised*. The MIT Press, 1997.
66. K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal of Software Tools for Technology Transfer*, 9(3):213–254, 2007.
67. N. Russell and A. H. M. ter Hofstede. Surmounting BPM challenges: the YAWL story. *Computer Science - R&D*, 23(2):67–79, 2009.
68. W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: yet another workflow language. *Information Systems*, 30(4):245–275, 2005.
69. W. M. P. van der Aalst, L. Aldred, M. Dumas, and ter Hofstede A. H. M. Design and Implementation of the YAWL System. In *16th International Conference on Advanced Information Systems Engineering (CAiSE'04)*, pages 142–159, 2004.
70. N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. newYAWL: Achieving Comprehensive Patterns Support in Workflow for the Control-Flow, Data and Resource Perspectives. Technical Report BPM-07-05, BPM Center Report, 2007. http://bpmcenter.org/wp-content/uploads/reports/2007/BPM-07-05.pdf. Accessed June 2012.
71. N. Russell, W. M. P. van der Aalst, and A. H. M. ter Hofstede. Designing a Workflow System Using Coloured Petri Nets. *Transactions on Petri Nets and Other Models of Concurrency*, 3:1–24, 2009.
72. Wil M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

73. OASIS Standard. *Web Services Business Process Execution Language Version 2.0*, April 2007. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.
74. E. Börgher. Approaches to modeling business processes. A critical analysis of BPMN, workflow patterns and YAWL. *Software and Systems Modeling*, 2011.
75. Wil M. P. van der Aalst. Verification of Workflow Nets. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets (ICATPN'97)*, pages 407–426, 1997.
76. W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, and M. T. Wynn. Soundness of workflow nets: classification, decidability, and analysis. *Form. Asp. Comput.*, 23(3):333–363, 2011.
77. Axel Martens. Usability of Web Services. In *International Conference on Web Information Systems Engineering Workshops*, pages 182–190, Los Alamitos, CA, USA, 2003. IEEE Computer Society.
78. Juliane Dehnert and Peter Rittgen. Relaxed Soundness of Business Processes. In *Proceedings of the 13th International Conference on Advanced Information Systems Engineering*, CAiSE '01, pages 157–170, London, UK, UK, 2001. Springer-Verlag.
79. Frank Puhlmann and Mathias Weske. Investigations on soundness regarding lazy activities. In *Proceedings of the 4th International Conference on Business Process Management (BPM'06)*, pages 145–160, Berlin, Heidelberg, 2006. Springer-Verlag.
80. A. Awad, G. Decker, and N. Lohmann. Diagnosing and Repairing Data Anomalies in Process Models. In *BPM Workshops*, 2009.
81. N. Lohmann. Correcting Deadlocking Service Choreographies Using a Simulation-Based Graph Edit Distance. In *6th International Conference on Business Process Management (BPM'08)*, pages 132–147, 2008.
82. A. Arcuri. On the automation of fixing software bugs. In *30th International Conference on Software Engineering (ICSE)*, pages 1003–1006, 2008.
83. S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *11th Annual Conference on Genetic and Evolutionary Computation (GECCO'09)*, pages 947–954, 2009.
84. Josh L. Wilkerson and Daniel Tauritz. Coevolutionary automated software correction. In *12th Annual Conference on Genetic and Evolutionary Computation (GECCO'10)*, pages 1391–1392, 2010.
85. B. Suman. Study of simulated annealing based algorithms for multiobjective optimization of a constrained problem. *Computers & Chemical Engineering*, 28(9), 2004.
86. K.I. Smith, R.M. Everson, J.E. Fieldsend, C. Murphy, and R. Misra. Dominance-Based Multiobjective Simulated Annealing. *IEEE Trans. on Evolutionary Computation*, 12(3), 2008.
87. Thomas Freytag. WoPeD - Workflow Petri Net Designer. In *Tool Demonstration. 26th International Conference On Application and Theory of Petri Nets and Other Models of Concurrency*, 2005.
88. M. T. Wynn, H. M. W. Verbeek, Aalst, Ter A. H. M. Hofstede, and D. Edmond. Business Process Verification - Finally a Reality! *Business Process Management Journal*, 15(1):74–92, 2009.
89. R.M. Karp and R.E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969.
90. N. Lohmann, E. Verbeek, and R.M. Dijkman. Petri Net Transformations for Business Processes – A Survey. *Transactions on Petri Nets and Other Models of Concurrency*, 2:46–63, 2009.
91. H. Bunke and G. Allermann. Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters*, 1(4):245–253, 1983.

92. Haiping Zha, Jianmin Wang, Lijie Wen, Chaokun Wang, and Jiaguang Sun. A workflow net similarity measure based on transition adjacency relations. *Computers in Industry*, 61(5):463–471, 2010.
93. Karsten Schmidt. LoLA: a low level analyser. In *Proceedings of the 21st international conference on Application and theory of petri nets*, ICATPN'00, pages 465–474, Berlin, Heidelberg, 2000. Springer-Verlag.
94. D. Fahland, C. Favre, B. Jobstmann, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf. Instantaneous Soundness Checking of Industrial Business Process Models. In *7th International Conference Business Process Management (BPM'09)*, pages 278–293, 2009.
95. Rong Liu and Akhil Kumar. An Analysis and Taxonomy of Unstructured Workflows. In *3rd Int. Conference Business Process Management (BPM'05)*, pages 268–284, 2005.
96. Ralf Laue and Jan Mendling. The Impact of Structuredness on Error Probability of Process Models. In *United Information Systems Conference (UNISCON), 2nd International Conference*, pages 585–590, April 2008.
97. H. Reijers and J. Mendling. Modularity in Process Models: Review and Effects. In *6th Int. Conference on Business Process Management (BPM'08)*, pages 20–35, 2008.
98. R. Laue and J. Mendling. Structuredness and Its Significance for Correctness of Process Models. *Information Systems and E-Business Management*, 8(3):287–307, 2009.
99. J. Vanhatalo, H. Völzer, and J. Koehler. The Refined Process Structure Tree. *Data & Knowledge Engineering*, 68(9):793–818, 2009.
100. Artem Polyvyanyy, Luciano García-Bañuelos, and Marlon Dumas. Structuring acyclic process models. *Information Systems*, 37(6):518–538, 2012.
101. Gul Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
102. Luc Bläser. A component language for structured parallel programming. In *Proceedings of the 7th joint conference on Modular Programming Languages (JMLC'06)*, pages 230–250. Springer-Verlag, 2006.
103. Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.
104. Shawn Bowers, Bertram Ludascher, Anne H. H. Ngu, and Terence Critchlow. Enabling Scientific Workflow Reuse through Structured Composition of Dataflow and Control-Flow. In *Proceedings of the 22nd International Conference on Data Engineering Workshops (ICDEW'06)*, pages 70–79, 2006.
105. Edward A. Lee and Thomas Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–799, 1995.
106. Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing Congress*, pages 471–475, 1974.
107. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
108. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The 3rd Edition*. Addison-Wesley Professional, 2005.
109. W3C Recommendation. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, November 2008. http://www.w3.org/TR/REC-xml/.
110. W3C Recommendation. *XQuery 1.0: An XML Query Language (Second Edition)*, January 2011. http://www.w3.org/TR/xquery/.
111. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

112. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
113. R. Strnisa, P. Sewell, and M. J. Parkinson. The Java Module System: Core Design and Semantic Definition. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 499–514, 2007.
114. G.M. Bierman, M.J. Parkinson, and A. M. Pitts. MJ: An Imperative Core Calculus for Java and Java with Effects. Technical report, University of Cambridge, 2003.
115. Rescher N. *Many Valued Logic*. McGraw-Hill, 1969.
116. Panti G. Multi-valued Logics. In *Handbook of Defensible Reasoning and Uncertainty Management Systems*, pages 25–74. 1998.
117. G. Oulsnam. Unravelling Unstructured Programs. *Computer Journal*, 25(3):379–387, 1982.
118. Sara Migliorini. *Supporting Distributed Geo-Processing: A Framework for Managing Multi-Accuracy Spatial Data*. PhD thesis, University Verona, Department of Computer Science, 2012.
119. Manfred Reichert. What BPM Technology Can Do for Healthcare Process Support. In *13th Conference on Artificial Intelligence in Medicine, (AIME'2011)*, pages 2–13, 2011.
120. Elliott M. Antman and et al. ACC/AHA Guidelines for the Management of Patients with ST-Elevation Myocardial Infarction. *Circulation*, 110(5):588–636, 2004.