Towards Structured Business Process Modeling Languages

Carlo Combi, Mauro Gambini, and Sara Migliorini

Department of Computer Science – University of Verona
Strada Le Grazie, 15, 37134 Verona, Italy
{carlo.combi|mauro.gambini|sara.migliorini}@univr.it

Abstract. A Process-Aware Information System (PAIS) is a software system driven by explicit Business Process (BP) models. A basic PAIS provides at least an execution engine, a Business Process Modeling Language (BPML) and a graphical editor. The editor is mainly used to design new BP models, maintain existing ones and check their correctness. BPMLs typically embrace an unstructured control-flow paradigm that allows one to create a BP model by connecting the available elements without following any specific construction pattern. Nevertheless, several research efforts confirm that it is a good practice to use structured forms whenever possible in order to avoid subtle errors. This paper aims to promote the design of novel BPMLs able to support a fully structured control-flow approach. This goal is pursued by exposing pitfalls and myths that surround unstructured BPMLs and prevent any further investigation in the structured direction. The desired features of a structured BPML are outlined at the end of the paper.

Key words: process-aware information systems, structured business process modeling languages, structured control-flow, modularity.

1 Introduction

Organizations typically need to coordinate the efforts of different agents in order to achieve their established goals. A set of interrelated activities, performed by a group of agents, may constitute a *business process* (BP) with a well-defined outcome. A *Process-Aware Information System* (PAIS) is a software system driven by explicit BP specifications with the aim to coordinate the involved agents in performing their activities [1]. A BP is usually described through a high-level graphical *Business Process Modeling Language* (BPML) often accompanied with a low-level executable specification. The latter is interpreted by the PAIS engine to enact the process. This solution can be found in YAWL [1], BPMN/WS-BPEL [2] and other widely adopted languages.

BPMLs are usually classified as *unstructured* due to their graph-oriented syntax and token-based semantics inspired by Petri Nets; they allow free composition of constructs without worrying much about type or position of the connected elements. Using an unstructured BPML one can build more or less

structured models: 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 [3, 4]. Research concerning the quality of BP models confirms that structured forms are preferable than unstructured ones, because they improve model comprehensibility and reduce the probability to accidentally introduce errors inside models [5, 6]. These effects should not surprise: firstly, the presence of a structured form is an easy-to-verify syntactic property that can guarantee the presence of desirable semantic properties which can be hard to prove in the general case; secondly, structured forms enhance modularity, a key property in any complex design activity performed by humans.

The aim of this paper is to promote the development of a new generation of BPMLs that recognizes modularity as a primary requirement. In particular, new modeling languages should at least be able to enforce a fully structured control-flow design without compromising expressiveness. Modularity of a BPML can be defined as the ability to decompose models in small interrelated components which in turns can be recombined in different configurations. Modularity is becoming a main concern in BP design because BP models are growing in size due to technological advances in process automation and integration.

The remainder of this paper is organized as follows. Sec. 2 briefly discusses several research efforts that are at the basis of the following work. Sec. 3 introduces the notion of executable unstructured BPML and the basic notations adopted through the entire paper. Sec. 4 describes the most severe errors that can be introduced in a BP model created with an unstructured BPML. An expert designer essentially adopts structured forms whenever possible to avoid this kind of errors. Nevertheless, unstructured compositions are accepted as a necessary evil because they cannot be removed in all situations or they are at least more convenient to use. Sec. 5 explains how this and other similar arguments are improperly used to exclude the existence of good structured BPMLs or to diminish their relevance. Sec. 6 shows how a structured BPML can be effectively built out of simple constructs and composition rules. Finally, Sec. 7 summarizes the findings and outlines future work.

2 Related Work

In [4] 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 [3] 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-featured structured BPMLs. In [5] Mendling and Laue investigate the importance of structuredness for obtaining correct models: they

introduce two different metrics that capture the degree of unstructuredness and related these metrics to the probability of finding an error. They conclude that structuredness is an important property for increasing the quality of BP models. Moreover, Reijers and Mendling in [6] analyze modularity and conclude that such property has positive effects on the comprehensibility of large-scale BP models.

These effects are also confirmed by Vanhatalo et al. in [7] where they present a parsing technique, called Refined Process Structure Tree (RPST), useful for detecting structured subgraphs inside a BP model. RPST has several applications, for instance it can be used as a first step for translating a graphical BP model into low-level executable specification which can be directly interpreted by the PAIS engine. A first analysis about modularity and concurrency of BPMLs can be found in [8], where we 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 a complete computational model.

3 Background

Many existing BPMLs can be classified as both executable and unstructured. It is considered *unstructured* any BPML that exposes a graph-oriented syntax with a token-based semantics rooted in the Petri Nets theory. A BPML is also classified as *executable* when it allows one to produce BP models containing all the information needed to be directly interpreted by a PAIS. An executable BPML can be distinguished from a pure conceptual one by the presence of dedicated constructs for declaring and manipulating data.

Without claiming to be exhaustive, the notion of executable unstructured BPML captures a substantial set of existing modeling languages. YAWL and BPMN, with its WS-BPEL executable semantics, are certainly good representatives of this category. YAWL [1] was born as an academic research project, it offers a clearly stated semantics and captures many of the workflow control-flow and data patterns in a coherent system. Furthermore, through the Workflow Pattern initiative, YAWL can be easily compared with other BPMLs. BPMN [2] is a widely adopted industrial standard that has reached a good acceptance and is supported by many commercial systems.

This paper contains some abstract BP models expressed in YAWL or BPMN. Whenever necessary, these models are augmented with additional ad-hoc graphical notations that are not natively provided by the chosen BPMLs. In particular, a task T that initially reads a set of variables $\bar{x} = \{x_i\}_{i=1}^{n}$ and subsequently writes a set of variables $\bar{y} = \{y_j\}_{j=1}^{m}$, not necessary disjoint, is graphically denoted as $[x_1, \ldots, x_n \mid T \mid y_1, \ldots, y_m]$, where the square brackets represents the graphical box boundary. The simplified notation $[x_1, \ldots, x_n \mid T]$, $[T \mid y_1, \ldots, y_m]$ or $[T]$ is used when $\bar{x} = \emptyset$, $\bar{y} = \emptyset$ or $\bar{x} = \bar{y} = \emptyset$, respectively. Input variables $\bar{x}$ are always placed on the side of the incoming control-flow arrow, while output variables $\bar{y}$ on the opposite side. Finally, YAWL cancellation regions are represented with dashed ellipses connected to the reset task through a line of the same type.

4 Common Pitfalls in Unstructured BP Modeling

Many kind of errors can be accidentally introduced in a BP 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 discarded due to a lack of expressiveness of the available BPMLs and they are commonly accepted as a necessary evil.

The abstract BP model in Fig. 1 will be used through the entire section to ease the discussion. The task A updates the variable x that is read by B and F; the task F also takes the value produced by E that is temporally 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 $\{u_1, A, s, B, v_1\}$. Task 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 the loop, task 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 as the one in Fig. 2, because it contains several issues. Each of them is used to introduce the general problems that arise with an unstructured control-flow design.

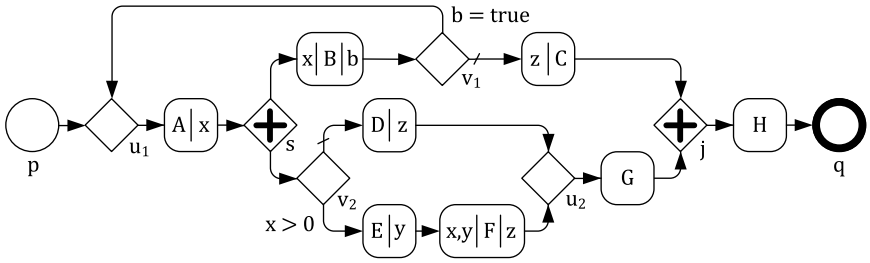

Fig. 1. An unstructured BP model containing some subtle errors. The model is expressed in BPMN and annotated with variable names and routing construct identifiers.

(P1) Control-flow entanglement – In many unstructured BPMLs 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. 1 there is no guarantee that C reads exactly the last value produced by D or F. Eventually, the following trace occurs: $p \rightarrow u_1 \rightarrow A \rightarrow A|x \rightarrow s \rightarrow (x|B, v_2) \xrightarrow{x \leq 0} (B, D) \rightarrow (B|b, D) \rightarrow (v_1, D) \xrightarrow{b=false} (z|C, D) \rightarrow (C, D) \rightarrow (j, D) \rightarrow (j, D|z) \rightarrow (j, u_2) \rightarrow (j, G) \rightarrow (j, j) \rightarrow H \rightarrow q$. 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 leaved in different places of model during its execution. For example, the model in Fig. 1 suffers of an improper completion because the loop $\{u_1, A, s, B, v_1\}$ potentially produces more than one token that flow through the bottom branch of the And-Split s, finishing their run in j. Indeed, 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 used for synchronization purpose.

(P3) Unreliable invariants – With an unstructured BPML one cannot exclude that multiple tokens flow in the same sequential branch making invariants hard to state and preserve. For instance, 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 BP model becomes soon a non-trivial activity. For example F in Fig. 1 belongs to a branch guarded by $x > 0$, but in this case one cannot exclude that F computes $z \leftarrow y/x$ with $x = 0$. Indeed, the following trace can occur: $p \to u_1 \to A \to A|x \to s \to (x|B, v_2) \xrightarrow{x>0} (B, E) \to (B|b, E) \to (v_1, E) \xrightarrow{b=true} (u_1, E) \to (A, E) \to (A|x, E) \xrightarrow{x=0} (s, E) \to (s, E|y) \to (s, \{x, y\}|F)$ leading to the erroneous computation $y/0$.

5 Myths Surrounding BPMLs

This section discusses some common misconceptions about BPMLs that seem to be so widespread to prevent any further investigation towards structured modeling approaches. The central concepts of the section are exemplified through the abstract model in Fig. 2. The model is extracted and adapted from [4], 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 $\{s_1, E, j_2\}$.

(M1) The myth of expressiveness – *Executable unstructured BPMLs are more expressive than structured ones in terms of definable BP models*. This myth is rooted on the elusive notion of structured BPML that lacks of an accepted reference implementation and it is also supported by mathematical proofs [4]. Actually, these proofs are very weak because they start defining a structured BPML as an unstructured BPML subject to further syntactical restrictions. Moreover, the BPMLs under analysis are simplified modeling languages that can hardly be considered executable; for instance, they lack of any construct to declare and manipulate data. These proofs usually shows that there exist well-behaved unstructured BP models, as the one in Fig. 2, that cannot be expressed with a structured control-flow. The used argument is fairly trivial because any language with additional constraints is likely to become less expressive, at least

if the constraints have some tangible effect. Moreover, it shall be assumed that the BPML at hand is not 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. However, interpretation is not necessary at all with the right set of constructs; for example, Sec. 6 presents a model behaviorally equivalent to the one in Fig. 2 but with a structured control-flow.

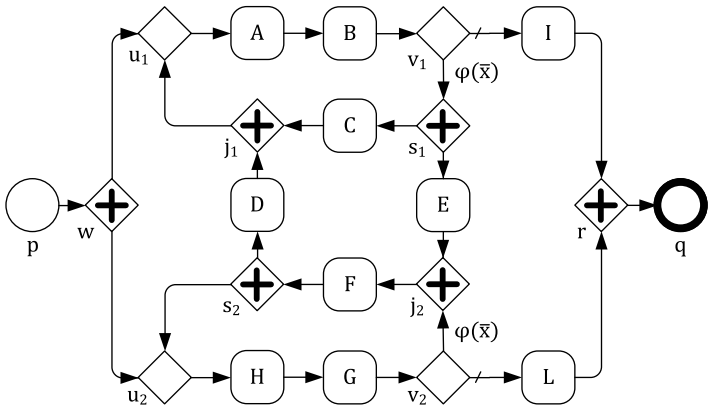

Fig. 2. A well-behaved unstructured BP model expressed in BPMN.

(M2) The myth of soundness – *Well-behaved BP models can be easily distinguished from erroneous ones using the available validation methods*. This myth is fostered by the progressive advances in verification methods [9] 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 BP 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, for instance an 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. 2 is sound with respect to the usual notion of soundness adopted for workflow nets [1]. Nevertheless, 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 $\bar{x}$ changes, there exists at least one task 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 $p \rightarrow w \rightarrow (u_1, u_2) \rightarrow (A, u_2) \rightarrow (A, H) \rightarrow (A, G) \rightarrow (A, v_2) \xrightarrow{\varphi(\bar{x})=true} (A, j_2) \rightarrow (A|x_i, j_2) \rightarrow (B, j_2) \rightarrow (v_1, j_2) \xrightarrow{\varphi(\bar{x})=false} (I, j_2) \rightarrow (r, j_2)$. In this case the passed soundness check may generate a false expectation about BP model correctness.

(M3) The myth of refactoring – *Any unstructured BP model can be refactored in a better one, not necessarily structured*. This myth is rooted on the questionable assumption that an unstructured BPML offers more design freedom because it imposes less construction rules. Actually, unstructured BPMLs 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 [8] for the model in Fig. 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 BP model is likely to grow in size as details are added and the creation of sub-processes, 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 BPMLs, probably because it cannot be easily integrated with the adopted language constructs. For instance, BPMN could describe the communication between R and Q improperly placing them into two different pools, but the main process P cannot be modeled anymore because message passing is prohibited in the same process model.

6 The NestFlow Modeling Language

This section introduces a novel modeling language called NestFlow [10], in order to show how a BPML can support a fully structured control-flow design approach without compromising expressiveness. The following introduction includes only a small set of simplified NestFlow constructs sufficient to understand the examples in Fig. 4 and Fig. 5. For a complete description of NestFlow the reader can refer to [10]. The essential NestFlow constructs and basic compositional rules are given

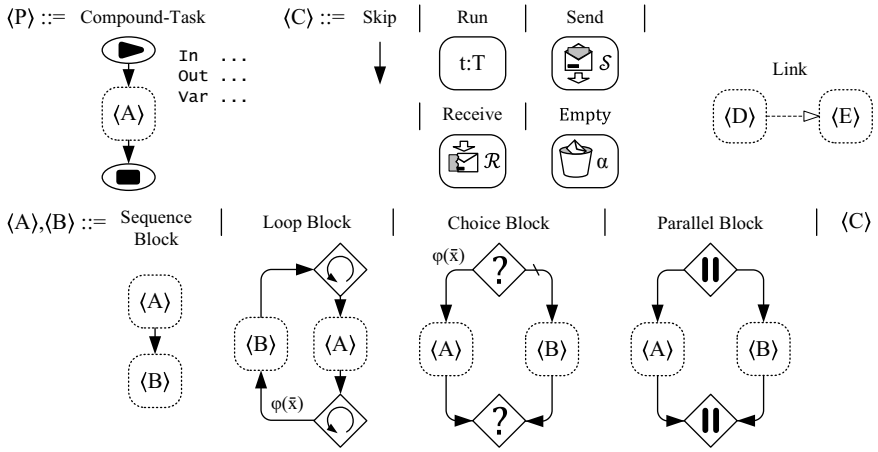

Fig. 3. The graphical syntax of the NestFlow modeling language.

in Fig. 3 through a graphical extension of the Backus-Naur-Form (BNF). In such figure, $\langle P \rangle$ denotes the starting symbol, | is the usual BNF choice operator, $\langle A \rangle$ and $\langle B \rangle$ denote non-terminal control-flow blocks and $\langle C \rangle$ denotes terminal

command blocks. A NestFlow model is obtained by expanding the body of the `compound-task` with the $\langle A \rangle$ and $\langle C \rangle$ elements. A `compound-task` declares zero or more local variables and a set of input and output streams that represent its external interface. A repeat-until loop and a while loop can be obtained specializing a generic `loop` block with a `skip` command placed in $\langle B \rangle$ or $\langle A \rangle$, respectively. For simplicity, `choice` and `parallel` blocks are limited here to only two branches; the corresponding generic constructs can be simulated by nesting multiple times these simpler blocks. A structured synchronizing merge can be implemented with one or more nested parallel blocks with a `choice` block for each parallel branch. A native task or a previously defined `compound-task` can be invoked inside a model with a `run`. A `send` copies the value of a variable inside an input stream, while a `receive` takes an object from an output stream storing it in a local variable. An output stream can be connected to an input stream by means of a *link*, graphically denoted with a dashed arrow as in Fig. 3: $\langle D \rangle$ can be a task or a `send` , similarly $\langle E \rangle$ can be another task or a `receive`.

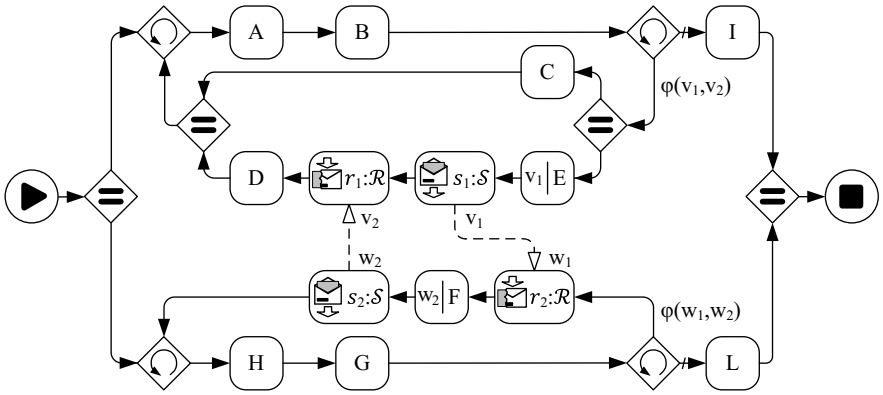

Fig. 4. A NestFlow BP model with a structured control-flow.

In Fig. 4 is depicted an abstract BP model expressed in NestFlow that resembles the one of Fig. 2. In contrast with traditional BPMLs, NestFlow promotes message passing as the primary abstraction for manipulating data: message passing is used to model the flow of data and objects exchanged by different processes as well as by internal tasks that belong to the same scope. Parameter passing can be easily stated in terms of message passing primitives, hence it should be intended as syntactic sugar in NestFlow; in particular, $[\bar{x}|T]$ represents a `send` command invoked before T, while $[T|\bar{y}]$ represents a `receive` command invoked after T. For instance in Fig. 4, $[E|v_1]$ can be represented as a message exchange between the invoked task instance $e:E$ and a `receive` placed immediately after E that stores the transmitted value in the local variable v_1.

The model in Fig. 4 has a structured control-flow and it can be considered behavioral equivalent to the unstructured one of Fig. 2, provided that the latter correctly synchronizes its internal state before evaluating the $\varphi(\bar{x})$ condition. In NestFlow parallel branches cannot share local variables, indeed it is clear that the

φ condition in Fig. 4 shall be checked over two different sets of variables $\{v_1, v_2\}$ and $\{w_1, w_2\}$. The correct synchronization of the loop conditions is guaranteed by the explicit links (s_1, r_2) and (s_2, r_1) that imply $w_1 = v_1$ and $v_2 = w_2$, hence $\varphi(v_1, v_2) = \varphi(w_1, w_2) = \varphi(v_1, w_2)$. For this reason, and assuming that variables are initially set to the same value, the model cannot reach a deadlock due to the φ predicate. The richer data-flow notation of NestFlow can effectively help in designing correct executable BP models since the beginning.

Moreover, the two loops in Fig. 4 can be easily encapsulated into two different components that mutually interact through a couple of links without exposing their internal implementation. As discussed in the previous section, this transformation is difficult, if not impossible, to obtain with traditional BPMLs due to their task activation semantics and unstructured control-flow.

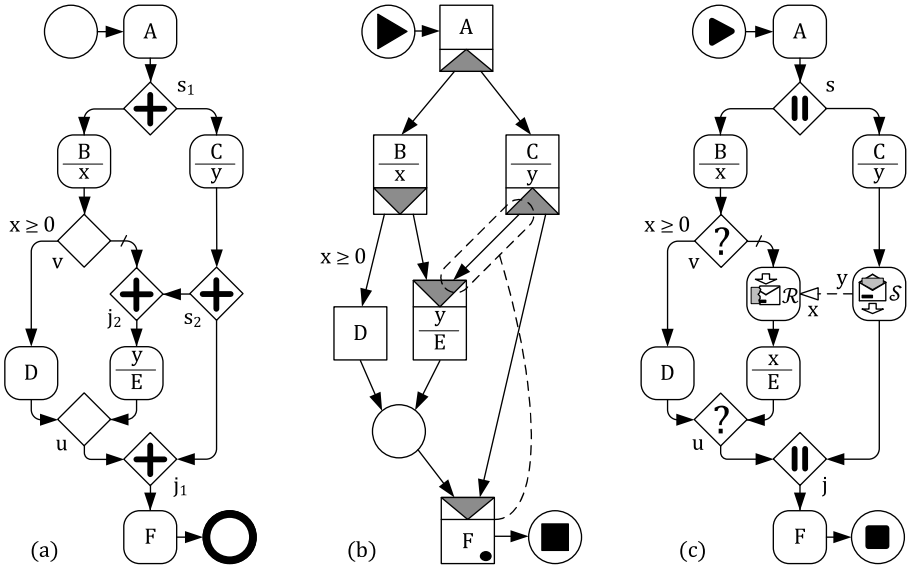

Fig. 5. The different roles of objects and threads in control-flow modeling.

Message passing is essential for enhancing modularity and effectively helps in unraveling complex control-flow relations. For this purpose, links can be considered flexible control-flow dependencies managed by a block-structured control-flow logic. The flexibility comes from the semantics of the `receive` command: a `receive` temporarily suspends the current thread of control that can be resumed when a timeout expires or by receiving an object from a different source.

In the abstract models of Fig. 5, tasks B and D execute in parallel with C, while E has to wait 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.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.b. In the NestFlow model of Fig. 5.c the dependency between C

and E is represented through a link that connects the two parallel branches. It is a reasonable solution as 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 with an optional `empty` command.

7 Conclusion

Research provides evidence that a structured control-flow design improves the quality of BP models reducing the presence of errors. Despite these results there is no serious attempt to investigate BPMLs able to support a complete structured control-flow design. This paper identifies the limits of an unstructured modeling approach and exposes the weaknesses of the principal arguments used to neglect the adoption of a structured BPML. Finally, some desired characteristics of a structured BPML are emphasized by means of two examples. These examples are expressed using a novel BPML, called NestFlow [10], which enforces a structured control-flow coupled with message passing constructs.

References

1. A.H.M. ter Hofstede, W.M.P. van der Aalst, M. Adams, and N. Russell. *Modern Business Process Automation: YAWL and its Support Environment*. 2009.
2. Object Management Group (OMG). *Business Process Modeling Notation (BPMN) 2.0 (Beta 1)*, August 2009. `http://www.omg.org/spec/BPMN/2.0/`.
3. R. Liu and A. Kumar. An Analysis and Taxonomy of Unstructured Workflows. In *Business Process Management (BPM), 3rd Int. Conference*, pages 268–284, 2005.
4. B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler. On Structured Workflow Modelling. In *Advanced Information Systems Engineering (CAiSE), 12th International Conference*, pages 431–445, 2000.
5. R. Laue and J. Mendling. The Impact of Structuredness on Error Probability of Process Models. In *United Information Systems Conference (UNISCON), 2nd International Conference*, pages 585–590, 2008.
6. H. Reijers and J. Mendling. Modularity in Process Models: Review and Effects. In *Business Process Management (BPM), 6th Int. Conference*, pages 20–35, 2008.
7. J. Vanhatalo, H. Völzer, and J. Koehler. The refined process structure tree. *Data & Knowledge Engineering*, 68(9):793 – 818, 2009. Sixth International Conference on Business Process Management (BPM 2008) - Five selected and extended papers.
8. C. Combi and M. Gambini. Flaws in the Flow: the Weakness of Unstructured Business Process Modeling Languages Dealing with Data. In *On the Move to Meaningful Internet Systems (OTM), 17th International Conference on Cooperative Information Systems (CoopIS)*, pages 42–59, 2009.
9. 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.
10. Carlo Combi, Mauro Gambini, and Sara Migliorini. The NestFlow Interpretation of Workflow Control-Flow Patterns. In *15th International Conference on Advances in Databases and Information Systems (ADBIS)*, 2011.