Flaws in the Flow: The Weakness of Unstructured Business Process Modeling Languages Dealing with Data

Carlo Combi and Mauro Gambini

Dipartimento di Informatica – Università di Verona
Strada Le Grazie, 15, Verona, Italy
{carlo.combi,mauro.gambini}@univr.it

Abstract. Process-Aware Information Systems (PAISs) need more flexibility for supporting complex and varying human activities. PAISs usually support business process design by means of graphical graph-oriented business process modeling languages (BPMLs) in conjunction with textual executable specifications. In this paper we discuss the flexibility of such BPMLs which are the main interface for users that need to change the behavior of PAISs. In particular, we show how common BPMLs features, that seem good when considered alone, have a negative impact on flexibility when they are combined together for providing a complete executable specification. A model has to be understood before being changed and a change is made only when the benefits outweigh the effort. Two main factors have a great impact on comprehensibility and ease of change: concurrency and modularity. We show why BPMLs usually offer a limited concurrency model and lack of modularity; finally we discuss how to overcome these problems.

Keywords: process-aware information systems, unstructured business process modeling languages, concurrency, modularity, refactoring.

1 Introduction

Business process modeling involves the understanding of the observed or the desired reality and the design of a consistent specification which can be used for driving a Process-Aware Information System (PAIS) [1] to streamline the business. The description of a business process has to be shared among all the stakeholders that participate in the design activity to reach an agreement on how the business works and how the information system will be implemented.

PAISs support business process design by means of graphical Business Process Modeling Languages (BPMLs) in conjunction with textual executable specifications that contain additional data necessary at run-time. These languages are usually graph-oriented and they allow multiple connections among different parts of the same model, for instance connecting the body of two distinct loops.

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part I, LNCS 5870, pp. 42–59, 2009.

The business process management community recognizes several limits of PAISs, in particular the lack of flexibility [2,3]: the demand for more flexible systems is not surprising, especially for supporting complex and varying human activities. Flexibility is a pervasive concept in the design of a PAIS and it can be related to several aspects, for instance it may refer to the deployment of new processes or the ability to change process instances at run-time or else the allocation of human resources. In this paper we discuss the flexibility of BPMLs which can be considered the main interface of PAISs: they are used by process designers for capturing business logic, by developers for implementing a working system and by common users, especially if they need to perform some changes during the enactment of a process.

A software system has to be comprehended before being changed, at least for avoiding unpredictable results. Only by understanding the consequences of a change we can estimate the effort needed to make it and a change is made only when the benefits outweigh the effort. For improving BPMLs flexibility we need to enhance comprehensibility and reduce the effort needed to change business process models. Two main intertwined factors have a great impact on comprehensibility and ease of change, namely *concurrency* and *modularity*. In this paper we will bring to light frequent design flaws of graph-oriented BPMLs that severely limiting their ability to cope with concurrency and modularity, producing negative impacts on PAISs flexibility. For this purpose we introduce a typical BPML supporting (1) unstructured control-flow design with token-based semantics, (2) hierarchical decomposition of tasks, (3) parameter passing among local task variables and (4) asynchronous message passing. We show that such flaws are not related to a specific language aspect but they arise when different features are put together for obtaining a complete executable specification. In particular, unstructured control-flow features mixed with shared task variables offer a limited concurrency model; we explain why asynchronous message passing is not a viable alternative to shared variables communication because in conflict with hierarchical decomposition; finally we argue that the mixing of these language abstractions reduce modularity, severely limiting the ability to express and change large business process models.

The remainder of this paper is organized as follows: in Section 2 we introduce some related work. In Section 3 we clarify some basic notions such as concurrency, comprehensibility and modularity in the context of executable BPMLs. In Section 4 we propose a core modeling language for capturing the main features of unstructured BPMLs. In Section 5 we show the limits of business process modeling based only on control-flow constructs and we discuss the issues that arise when data are not carefully modeled during design. In Section 6 we show why asynchronous message passing cannot be used instead of shared data communication among tasks. In Section 7 we point out the difficulties to overcome for transforming a model preserving its semantics. In Section 8 we discuss plausible solutions for the exposed issues and we explain why there are no quick fixes. Finally, in Section 9 we summarise the results of this paper and we discuss open problems.

2 Related Work

Several interesting techniques have been proposed to gain flexibility in PAISs, for instance exception handling can be used for modeling unpredictable events [4]; the adaptability of business process models can be improved by supporting change patterns [5] or by allowing the composition of blocks starting from simple fragments [6]; data-driven and declarative approaches can be used to adapt the structure of a model at run-time [7,8].

At the best of our knowledge, the solutions to gain flexibility in PAISs fall into two categories: (1) solutions based on innovative techniques that go beyond the current practice like declarative ones and (2) solutions that improve existing systems and modeling languages by adding new features; these solutions are often related to a particular aspect of PAISs such as control-flow design. On the contrary this paper focuses on the interaction among common core features of existing BPMLs that are put together for providing a complete executable model. We will show that such mixing of features often goes against well-established software engineering principles like modularity and program comprehensibility [9,10] and hence against flexibility.

A business process model may be sound or not depending on the chosen soundness criteria. Many soundness criteria have been proposed, each one attempts to capture good models without being too restrictive [11]. Such criteria are usually defined through Petri Net Theory ignoring data-flow aspects, even though such aspects are relevant for obtaining executable specifications. The aim of this paper is not to define yet another soundness criteria but to explain some fundamental flaws in the design of BPMLs. For such purpose we use simple business process models that can be sound respect to particular criteria but are poor from a program comprehension perspective. These models are expressed using an abstract BPML for capturing the core features of common modeling languages; a broad comparison among features of existing BPMLs has been already done through the workflow patterns initiative [12,13].

3 Background

We define the data-flow of a system as the set of connections among components needed to exchange data. In a process we can distinguish different aspects and study them nearly independently; such aspects are called *perspectives* and mainly concern control-flow logic, data management and resource management [1]. Such distinction cannot be freely transferred to an executable BPML, because control-flow and data-flow concerns cannot be easily separated: we cannot ignore data dependencies if we would obtain correct specifications [14]. Some BPMLs neglect data-flow aspects, but their specification is only postponed when processes are implemented in a PAIS. For these reasons we consider jointly control-flow and data-flow aspects when we will discuss about BPMLs.

As mentioned in Section 1, for improving flexibility we have to enhance comprehensibility. Comprehensibility is a property of a system to be easily grasped

in all of its parts. Comprehensibility may refer to different subjects such as business processes, business process models or BPMLs; it is necessary to clarify these distinct but interrelated meanings: for a human, understanding a process means building a clear *mental* model about how it works. To understand a business process and share knowledge about it, we use a *concrete* model expressed in a certain BPML: a tool for tackling the inherent complexity of the observed reality. Therefore, clear semantics and comprehensibility are important requirements for BPMLs, but to be useful they must be able to manage complexity. As stated previously, two main intertwined factors have a great impact on comprehensibility of BPMLs, namely concurrency and modularity.

Reasoning about concurrent entities is difficult: for instance, nontrivial multithreaded programs become soon incomprehensible to humans [15]. Zapf et al. argue that it is not trivial to increase business process productivity by exploiting parallel tasks, because the needed coordination efforts reduce expected efficiency gains [16]. Nevertheless, parallelism cannot be avoided by removing it from models, since the modeled reality is inherently concurrent: not surprisingly one goal of a PAIS is to reduce coordination efforts into a truly concurrent environment, where agents need to interact with each others. Graph-oriented BPMLs adopt a token-based semantics which is typically formalized through Petri Nets Theory. Petri Nets are good for describing true concurrency: unfortunately the concept of token does not differ from the concept of thread when it is associated with the wrong data model. Not surprisingly, current systems impose properties like safeness [12] or they adopt structured control-flow constructs when possible.

Modularity is one of the few means to face complexity. We define modularity of a BPML as the ability to decompose its models in small interrelated components which in turns can be recombined in different configurations [9]. The lack of modularity makes large models hard to understand and change, since a slight modification in one part may affect the model as a whole with the risk to introduce new errors [9]. Modularity enhances the ability to change a model; anyway it is worth noting that adding features like hierarchical decomposition of tasks to a BPML does not necessary increase modularity: a large model decomposed into some compound tasks may be more comprehensible but more hard to change. Quantifying the modularity of a language is not a simple matter because we need to compare the size of each change with the efforts necessary to apply it. Nevertheless, the effort to make a change in a model must be proportional to the semantics gap between the current and the new version: as a consequence, small semantics changes should be easy to perform and semantics-preserving ones should be effortless, as for the creation of a sub-process in order to reduce the size of the original model and to enhance comprehensibility.

4 A Typical Unstructured Graph-Oriented BPML

In this section we introduce a typical graph-oriented BPML which supports the definition of unstructured models with multiple connections among components; in the following we will refer to it as $\mathcal{A}$ language.

Adopting the approach in [17], $\mathcal{A}$ is a *model* of existing systems like YAWL [18], BPMN with WS-BPEL executable semantics [19], and jBPM from the JBoss Enterprise Middleware [20] which resembles UML Activity Diagrams [21]. The aim is twofold: first, we gain generality by capturing relevant aspects of graph-oriented languages without restricting to a specific implementation and second, we base the discussion on a language with a well-defined behavior, since several languages propose their own particular constructs and expose a slightly different semantics often given only informally.

As the name suggests, the syntax of graph-oriented languages can be described with the Graph Theory, while its semantics can be given through the Petri Nets Theory. The formalization of a complete executable modeling language is not a simple matter: for our purposes we need only core constructs which were already analyzed in literature through the YAWL initiative, workflow control-flow and data-flow patterns [22,13,12].

The graphical symbols of the language are depicted in Fig. 1, whereas the main elements are summarized in the UML Class Diagram depicted in Fig. 2, where the in-degree and out-degree of each element are expressed within square brackets for not cluttering the diagram.

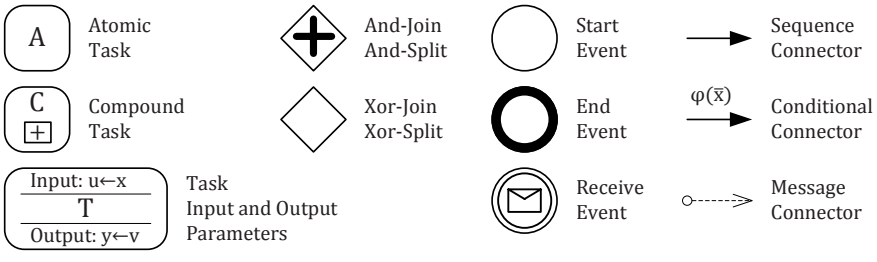

Fig. 1. BPMN symbols for $\mathcal{A}$ language constructs

Symbols in Fig. 1 are taken from the BPMN standard [19] except for the representation of input and output parameters of a generic task T, where x and y are external task variables, u and v are internal ones and the left arrow denote an assignment. When the internal implementation of a task is not relevant, we will denote only external variables affected by such task and whenever input and output are evident from control-flow path, we will drop the initial labels. A set of variables is denoted with a vector like $\overline{x}$, while a boolean expression over a set of variables $\overline{x}$ takes the form $\varphi(\overline{x})$. Two examples of a model expressed in $\mathcal{A}$ are depicted in Fig. 3 and Fig. 4.

For simplifying the language, out-degree of split gateways and in-degree of join gateways are limited to two: such limits do not compromise the freedom in the use of connectors, neither the generality of results. Control elements with higher degree are obtainable by composing basic constructs: for instance, inclusive Or-Split Or-Join pair can be simulated by And-Split followed by a Xor-Split for each parallel branch; this also holds for tasks and events with multiple entering

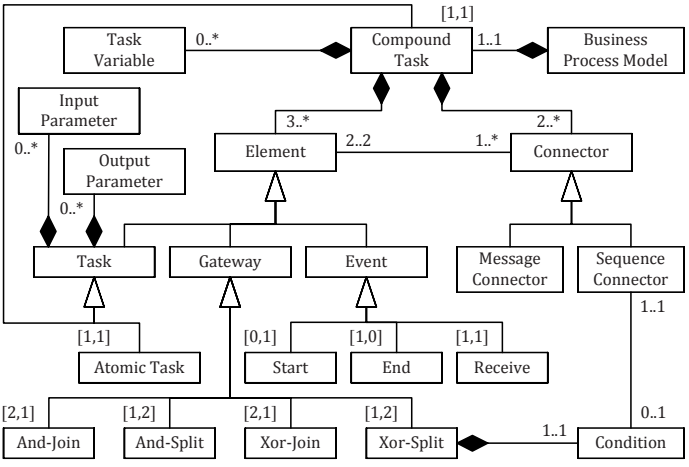

Fig. 2. The elements of the $\mathcal{A}$ modeling language

and leaving connectors. Arbitrary loops with many entry and exit points can be obtained with the available constructs.

The $\mathcal{A}$ language exposes a token-based semantics typical of unstructured BPMLs: a token is produced by a start event and consumed by an end event. Any model expressed in $\mathcal{A}$ could have more start and end events; we assume that all start events begin together as if they were connected backwards with one or more subsequent And-Splits. During the execution of an instance of the model, namely a *case*, every start event produces a token that follows the control-flow net defined by tasks, sequence connectors and gateways control elements. In particular, an And-Split construct consumes one token from the incoming branch and produces a new token for every outgoing branch, while the And-Join waits one token for each incoming branch before producing one outgoing token. With only two outgoing branches, a Xor-Split does not differ from an if-then-else construct, while the Xor-Join is interpreted as a simple merge [12] which replicates to the outgoing branch all the incoming tokens without synchronization. Every final event consumes tokens that arrive to it by removing them from the net and when all tokens have been removed the case terminates according to the implicit termination control-flow pattern [12]. When a token reaches a task, a new instance is enabled and loaded in the work-list ready to be performed; when the corresponding activity is finished, the task is closed and the token is released through the outgoing connector. This also holds for compound tasks, with the difference that a token is placed in every contained start event and it terminates when all the contained tokens have been removed.

Any compound task can declare one or more *task variables* of a certain type and the whole set of variables of a task instance is called *store*. A task instance is considered *stateless* because its store is not retained after the task conclusion. The scope of a variable coincides with the compound task where it is declared

and its visibility does not extend to task instances invoked in the same scope: in this manner $\mathcal{A}$ implements a form of encapsulation where each instance has its own local variables. The language does not support explicit termination [12], since it causes many troubles in concurrent context: a single end event may interrupt all parallel tasks possible leaving the stores into an inconsistent state.

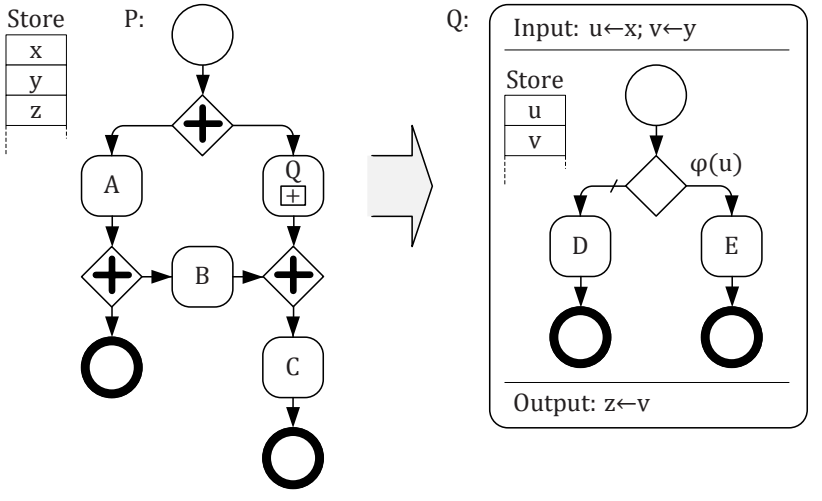

Fig. 3. Local Shared Variables (LSV) communication between tasks

A task can declare *input parameters* and *output parameters*; when a task is enabled, it reads the variables from the current scope and copy their content into its internal variables; when the task has completed, external variables are updated by means of output parameters as shown in Fig. 3. Indeed, from a data-flow perspective, any task T can be seen as a function between its input and output parameters, therefore we denote with $T : \overline{x} \mapsto \overline{y}$ such relation, where $\overline{x}$ is the set of variables read by T and $\overline{y}$ is the set of variables updated by T; for instance the task Q in Fig. 3 is denoted with $Q : \{x, y\} \mapsto \{z\}$. Moreover, we abstract from the computational aspects of the language, since any needed function $f : \overline{x} \mapsto \overline{y}$ can be implemented by an atomic task F with the aid of external programs or human intervention.

Tasks, even in a different scope, can communicate among them through input and output parameters and shared variables without breaking encapsulation. In the following, we will refer to this type of exchange as *Local Shared Variables* (LSV) communication. Beyond LSV, $\mathcal{A}$ supports *Asynchronous Message Passing* (AMP) communication for representing interactions among tasks. For sake of simplicity, we consider only point-to-point message exchanges from atomic send tasks to intermediate receive events: a message exchange between generic tasks means that the source task contains at least one atomic send task and the destination task contains at least one receive intermediate event. A receive intermediate event is implemented as message queue stored in the same scope

in which it is placed. As will become clear in the next sections, LSV remain the first choice for inter-task communication because AMP cannot be used in all situations.

5 Tokens and Shared Variables

In this section we expose the problems of LSV into a truly concurrent environment and the difficulty to take advantage of state invariants in presence of token-based semantics.

5.1 Control-Flow Design Hides Race Conditions

If we consider only control-flow concerns during the design of a business process, we may obtain a perfectly sound but unusable model, especially if such model takes advantage of parallelism. Sooner or later data-flow concerns must be taken into account and they could be in contrast with control-flow logic.

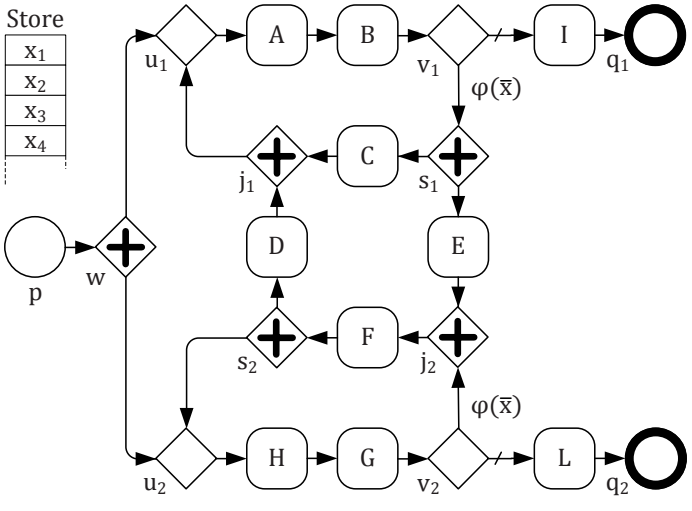

Fig. 4. An unstructured model with LSV communication strategy

For instance, let us consider the use of conditions in Xor-Split constructs using LSV communication strategy among tasks and in particular let us examine the use of loop conditions $\varphi(\overline{x})$ in the model depicted in Fig. 4. The model was used in [23] for proving the existence of well-behaved unstructured models that do not have a structured form. The model contains two main 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 and synchronize each other at every iteration. Abstracting from data-flow details, the model is free from deadlocks, but with LSV strategy it almost certainly reaches

an invalid state due to race conditions: there is no guarantee that the two conditions $\varphi(\overline{x})$ in the two loops are evaluated at the same time, so one loop may exit before the other.

We assume that the state underlying $\overline{x}$ changes during the execution, otherwise we would fall into two trivial cases: $\varphi(\overline{x})$ is always *true* and the process never terminates or $\varphi(\overline{x})$ is always $false$ and the loops are never executed entirely. Supposing that initially $\varphi(\overline{x})$ evaluates to *true*, if the state is mutable then there exists at least one task t that modifies one or more variables of the subset $\overline{x}$ by means of output parameters. For sake of simplicity we assume that exactly one task modifies the state, thereby for each $t \in \{A, \ldots, H\}$ excluding F we can exhibit an execution that lead to deadlock. Tasks I and L are not considered since they are placed outside the loops. To become convinced of this fact let us note that one of the two loops does not require to wait the end of t for reaching its Xor-Split gateway. For instance, let $t = A$, the trace $p \to w \to (u_1, u_2) \to (A, H) \to (A, G) \to (A, v_2) \to (A, j_2) \to (B, j_2) \to (v_1, j_2) \to (I, j_2) \to (q_1, j_2)$ leads to deadlock, in similar way a deadlock may occur for $t \in \{B, H, G\}$. For $t \in \{C, E\}$ the model may reach a deadlock in j_1 and j_2 respectively, since the branch terminating in G may still running when C and E finish. For $t = D$ the system can evolve from s_2 in $s_2 \to (u_2, D) \to (H, D) \to (G, D) \to (v_2, D) \to (j_2, D) \to (j_2, j_1) \to (j_2, u_1) \to (j_2, A) \to (j_2, B) \to (j_2, v_1) \to (j_2, I) \to (j_2, q_1)$. On the contrary, if the change of the state is made only by the task F the process does not run into a deadlock because F is correctly synchronized between the two loops.

5.2 Tokens Invalidate State Invariants

In languages like $\mathcal{A}$ we cannot rely on state invariants about a portion of the model to express business logic, as after a conditional gateway or inside the body of a loop: potentially, any assertion about a set of variables can be invalidated by a task instance enabled by another token that flows in the same branch.

Let us consider the model in Fig. 5. At least one task between A and B updates the state underlying $\overline{x}$, otherwise $\varphi(\overline{x})$ is always constant and the choice u is useless; if there exists an update, we can state that $\overline{x}_A \cup \overline{x}_B \neq \emptyset$, where $\overline{x}_A$ and $\overline{x}_B$ denote the subsets of variables of $\overline{x}$ modified by tasks A and B, respectively.

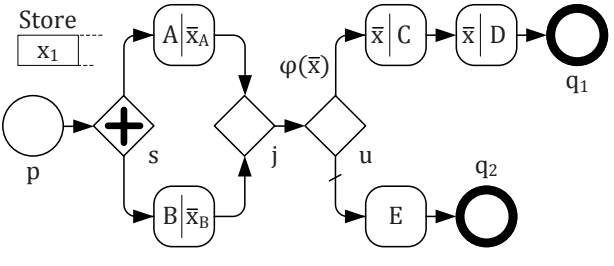

Fig. 5. Tasks C and D cannot rely on state invariant over $\overline{x}$

We discard trivial race conditions supposing $\overline{x}_A \cap \overline{x}_B = \emptyset$, so A and B tasks update different variables; we also suppose that $\varphi(\overline{x})$ is evaluated differently when the two tokens produced by the And-Split s reach the Xor-Split u, so we exclude concurrent activations of the same outgoing branch.

When A completes, a token reaches the Xor-Split u; if $\varphi(\overline{x})$ is true, C is enabled and reads the variables $\overline{x}$. At this point, if the state underlying $\overline{x}$ is updated by B before the conclusion of C, the task D will read different values than C such that $\varphi(\overline{x})$ is false. As a consequence, D cannot rely on the state invariant asserted by $\varphi(\overline{x})$ even if it is below a branch guarded by such assertion. Similar considerations hold if B finishes before A and in special cases when $\overline{x}_A = \emptyset$ or $\overline{x}_B = \emptyset$.

For instance, A and B may be activities counting the number of items into two warehouses $\mathcal{W}_A$ and $\mathcal{W}_B$, where the result is stored in $\overline{x}_A = \{x_1\}$ and $\overline{x}_B = \{x_2\}$ variables. Any time a counting activity finishes, the condition $x_1 \geq x_2$ is evaluated for deciding in which warehouse there are more items. If $x_1 \geq x_2$ the system schedules the task C which sends a notice by mail and then it enables task D which reads x_1 and x_2 to move $(x_1 - x_2)/2$ items from $\mathcal{W}_A$ to $\mathcal{W}_B$. When D is enabled, x_2 may have been already updated by B and so the computation $(x_1 - x_2)/2$ performed by D could be wrong.

6 Message Receivers Are Ambiguous

In this section we will explain why LSV cannot be easily replaced by AMP strategy. We have shown that LSV is not a good communication strategy, so one could ask why the AMP facility offered by $\mathcal{A}$ cannot be exploited in concurrent situations. Unfortunately AMP strategy is not a viable alternative for inter-task communication due to two main shortcomings: first, a task instance is stateless and it does not exist before its activation; second, the model does not guarantee that such instance is unique, because multiple instances may be enabled at the same time with a different dynamically assigned identifier. Thus, from a model it is not clear who is the receiver of a message, unless it is a specific intermediate event into an already enabled task. For example, in BPMN messages are used for modeling the interactions among instances of different process models, while within the same process, AMP among tasks like in Fig. 6 is forbidden in favour of strategies like parameter passing: in this manner BPMN guarantees that there exists only one process instance that receives a message.

To clarify the problem let us consider the model depicted in Fig. 6, where P_1 is the main process and P_2 is a compound task used by the first one. Due to the loop (v_1, s_1, A, B, u_1), P_1 can produce an unbounded number of tokens that flow through the right branch of s_1, so it can invoke P_2 multiple times. We suppose that the tasks of P_2 are highly interrelated and they need to exchange data using LSV strategy; as a consequence we have made P_2 an isolated compound task with its local store, so each instance of P_2 in the same case runs without interfere each other. An instance of P_2 can terminate only if e_2 receives a message from the external process P_1, that in turn guarantees the existence of one message for

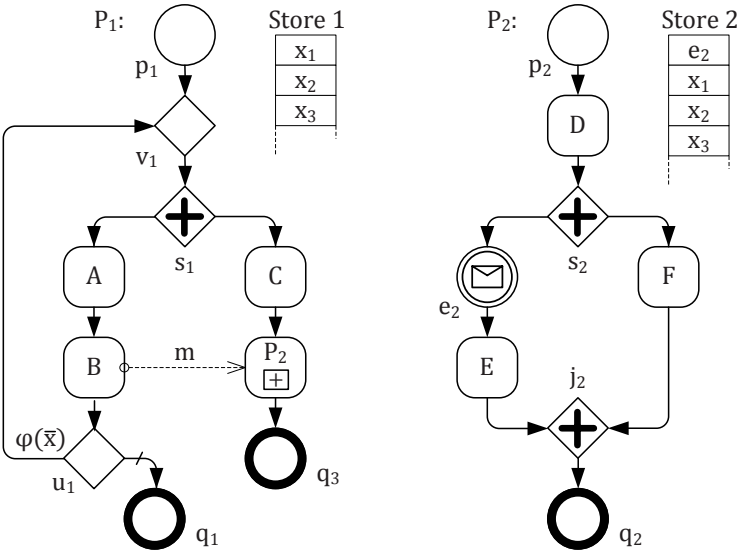

Fig. 6. An ill-defined message passing interaction

each instance, because they are started by the same loop (v_1, s_1, A, B, u_1) which afterwards produces a message m with the B send task.

The simplicity of the model hides a really complex execution semantics. First, there is no guarantee that an instance of P_2 exists when the first message m is sent, so intermediate event queues like e_2 are not sufficient to store the messages. As a consequence an external buffering strategy is needed and it cannot be as simple as a message queue: due to AMP semantics, the loop (v_1, s_1, A, B, u_1) may run multiple times and exit before a single instance of P_2 is created. The scheduling of these instances cannot be predicted because it depends on the order with which C instances are performed. As a result, to maintain the correct association between messages and receivers, the use of a message queue for each communication channel is not sufficient.

7 Unstructured Models and Language Modularity

In this section we discuss the modularity of unstructured BPMLs, in particular we point out which obstacles have to be overcome in order to make a common refactoring transformation useful for reducing the complexity of a model and gain comprehensibility.

7.1 Single Parts Are Complex as the Whole

We have argued that in executable models control-flow and data-flow concerns cannot be easily separated and we have pointed out the problems of LSV and AMP communication techniques in languages like $\mathcal{A}$. A different separation can

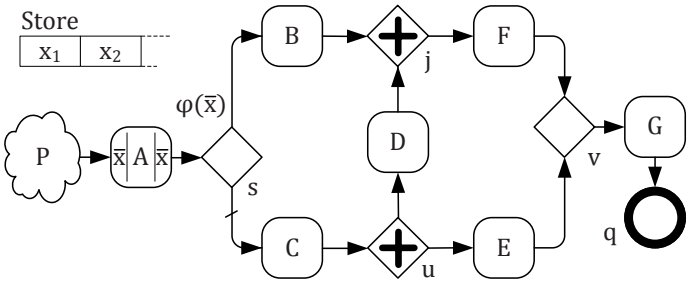

Fig. 7. An unstructured model with a first order improper nesting

be performed by isolating a portion of a model where a particular property holds; unfortunately, it is really difficult to state local properties in unstructured models due to arbitrary control-flow connections and token-based semantics.

Let us consider the model depicted in Fig. 7 and suppose for a moment that the sub-graph P is a single start event. The resulting model exhibits a first order improper nesting [24], because (s, j) and (u, v) are both corresponding pair of constructs that differ in their type and these pairs are not properly nested. This construction is considered erroneous since it leads to a deadlock for each of the two possible executions, because only one token can reach j in both cases.

In general, the correctness of the same pattern placed into an existing model cannot be determined without understanding the other parts. For instance, we reconsider the model in Fig. 7 but now preceded by an arbitrary graph P, and we suppose for simplicity that every time A is executed, the variables in $\overline{x}$ are updated to obtain a different evaluation of $\varphi(\overline{x})$, so the branches of the Xor-Split s are chosen alternatively: the A task can obtain this effect by reading the previous value of $\overline{x}$. If the number of tokens that exit from the sub-graph P is even, then this part of the model is free from deadlocks. The evenness of tokens exiting from P is a property of the entire graph: to infer something about a small part of the model one must consider the graph as a whole; even worse, the number of tokens into a graph is a property hard to derive statically from the model because it may vary in each case.

7.2 Chosen Constructs Kill Modularity

We have pointed out that one of the main goals of a model is to face the inherent complexity of the observed reality and we have stated that modularity is an essential property for managing complexity. In this section we argue that languages like $\mathcal{A}$ lack of modularity. In particular we reason about the difficulties encountered when we want to extract some compound tasks from a model and recombine them together to obtain a new model semantically equivalent to the original one: such transformation does not alter the semantics of the model but only the way in which it is represented for enhancing comprehensibility, therefore it must be effortless.

Let us reconsider the unstructured model in Fig. 4; because it is not so clear, we want to transform it in the model of Fig. 8, where P and Q are compound tasks that encapsulated the two main loops $(u_1, A, B, v_1, s_1, C, j_1)$ and $(u_2, H, G, v_2, j_2, F, s_2)$; moreover, we choose to place E in P and D in Q, obtaining two sub-graphs composed with $\{A, B, C, E\}$ and $\{H, G, F, D\}$. The details of this decomposition are reported in Fig. 9.

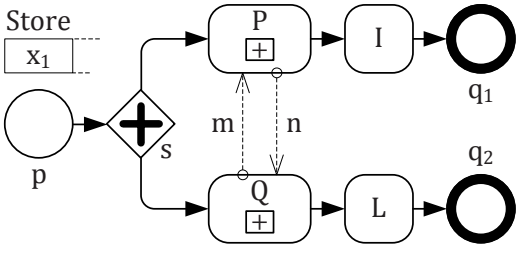

Fig. 8. A hierarchical decomposition between two interrelated tasks

A distinctive feature of unstructured BPMLs is allowing arbitrary connections among tasks; for this reason any control-flow sub-graph may have multiple entry and exit points. This feature, worsened by permissive token-based semantics, makes the extraction of a sub-graph a nontrivial operation. The control-flow dependencies among tasks cannot be discarded along with language constructs, so they must be kept by adding different constructs.

In the model of Fig. 4 only two main tokens can flow, one for each loop, while the tokens that flow through D and E are used for synchronization purpose, then it is fairly simple to remove crossing control-flow constructs. Direct control-flow dependences are given by $E \rightarrow F$ and $D \rightarrow A$, which can be replaced with two dummy messages m and n depicted in Fig. 8. We have just discussed the problems of AMP communication strategy: fortunately, in this example the main model assures the existence of one and only one receiver for each dummy message.

After the creation of a new compound task from a pre-existing control-flow sub-graph, data-flow relations must be adjusted since each compound task has its own local variables. This operation is nothing but trivial: on one hand, it is necessary to understand the order in which shared variables are updated and on the other hand, parameter passing must be replaced with message exchanges. For instance, let us consider the models in Fig. 4 and Fig. 8, and suppose that B updates a variable x_1 used by F: this is conceivable because B runs always before F for the control-flow dependencies $B \rightarrow E \rightarrow F$; for the same reason, let us suppose that D updates a variable x_2 used by A. In short, we can state $B : \emptyset \mapsto \{x_1\}$, $F : \{x_1\} \mapsto \emptyset$, $D : \emptyset \mapsto \{x_2\}$ and $A : \{x_2\} \mapsto \emptyset$. If we describe such relations through LSV parameter passing we have $P : \{x_2\} \mapsto \{x_1\}$ and $Q : \{x_1\} \mapsto \{x_2\}$, therefore these two compound tasks cannot run in parallel neither in sequence due to data-flow dependencies. As a consequence, AMP strategy must be adopted instead of LSV parameter passing which in turn does not have a clear semantics because of stateless tasks and multiple instances.

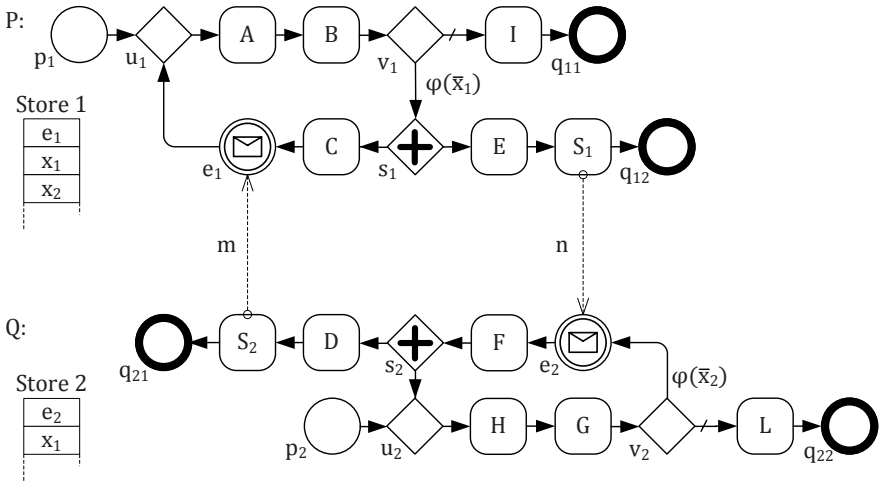

Fig. 9. Details of the hierarchical decomposition

This kind of transformations is common during the design and maintenance of a business process model whose size grows naturally for adjusting new requirements and special cases. Altogether the language abstractions chosen by languages like $\mathcal{A}$ for modeling business processes make these changes hard to perform.

8 Improving BPMLs

The primary aim of this paper is to clarify the interaction between control-flow and data-flow features that often undermine unstructured BPMLs flexibility: understanding these shortcomings is a required step for designing better BPMLs. In this section we summarize the obtained results and we discuss open problems and some potential solutions.

In the current practice data-flow aspects are usually ignored during business process modeling; as a consequence, in order to provide sufficient expressiveness, BPMLs need to support unstructured control-flow design with arbitrary connections among tasks [23]. However, data-flow concerns have to be considered for obtaining executable models able to drive a PAIS.

For existing BPMLs, the use of some verification methods seems the only viable solution for ensuring the absence of errors in the models or at least for warning about strange conditions. These methods may operate simultaneously on control-flow and data-flow relations to exclude common errors [14], however they may not be feasible on a full featured BPML, because detecting race conditions is an NP-Hard problem for any language able to express mutual exclusion [25]. In any case, whatever is the verification method used, ill-defined models have to be adjusted with the risk to introduce new errors or to lose the alignment between high level business logic and low level specification.

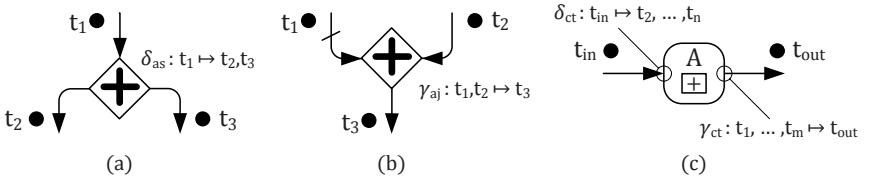

Fig. 10. Dispatch and combine extension points for TLV

These reasons justify the research of better data models for designing more flexible unstructured BPMLs that take care of concurrency and modularity issues. On one hand, LSV strategy offers a limited concurrency model and cannot scale for expressing complex business processes without AMP. On the other hand, AMP cannot be used in all situations, due to the stateless nature of tasks.

One possible solution is using a local store for each token, instead of a single store for each task. In this context a task receives tokens of a certain type, reads and updates all associated variables and releases them when completed. This behaviour is similar to parameter passing used in LSV strategy, but it does not have side effects due to shared variables; we refer to this strategy as Token Local Variables (TLV) communication.

In TLV every token has a unique identifier and its type provides information about the declared variables. Tokens are dynamically created by start event places or And-Split constructs, and they can be initialized using extension points as described in the following. Extension points are essentially functions for transferring data among tokens and they can be used for dispatching the data of one token to multiple ones, or for combining the data of many tokens to a single one. TLV requires advanced And-Split and And-Join constructs for supporting the extension points mentioned above:

- An And-Split has to support a dispatch extension point δ_{as} that receives the data of the incoming token t_1 and transforms them for producing the data associated to each outgoing token t_2 and t_3, as depicted in Fig. 10.a. By default, this extension point simply replicates the incoming data on each outgoing branch. Data may be only logically replicated through a lazy strategy, as they need to be physically copied only during an update.
- An And-Join has to support a combine extension point γ_{aj} that integrates the data of the two incoming tokens t_1, t_2 and resolves possible conflicts for producing the new outgoing token t_3, as in Fig. 10.b. If no combine function is provided, only the token that arrives from the default incoming branch is replicated in output after join. The default incoming branch can be represented as in Fig. 10.b.
- A compound task may have multiples start or end events; we have explained in Section 4 that all start events may be considered as connected backwards by an And-Split. Therefore, the creation of inner tokens does not differ from the And-Join semantics explained above, except for the use of a generalized function δ_{ct} that produces several tokens $t_1, \ldots, t_n$ from the incoming one t_{in}. Tokens $t_1, \ldots, t_m$ that reach end events are collected and when the compound

task is completed an extension point γ_{ct} is used for producing the outgoing token t_{out}, as in Fig. 10.c.

The TLV communication strategy can be a good data model for executable specifications with unstructured control-flow, at least for avoiding common concurrency issues. However, for modeling purposes, TLV presents some of the limits of LSV strategy: TLV associates data with tokens that are dynamically generated during the execution and are not statically visible from the model. Moreover, TLV strategy does not substantially improve the modularity of the language which largely depends on control-flow constructs.

9 Conclusions

In this paper we have introduced a typical modeling language, named $\mathcal{A}$, for capturing relevant features of existing graph-oriented BPMLs and we have considered jointly control-flow and data-flow concerns to have a broad perspective on modeling and execution issues.

We have argued that the essential features of languages like $\mathcal{A}$ do not fit well together, undermining their ability to express complex business process logic. Therefore, we have discussed some fundamentals language flaws related to (1) control-flow design into LSV context, (2) state invariants in presence of permissive token-based semantics, (3) the impedance between stateless tasks and AMP communication strategy, (4) the difficulty to verify a portion of a model isolated from the remaining ones and (5) the lack of modularity. Finally, we have considered the possibility to use a TLV strategy, where each token has its own local store. This solution offer a better data model for unstructured BPMLs, since it prevents some common race conditions. Unfortunately, it does not resolve all the problems highlighted for LSV and AMP strategies, because some of these are directly related to the choice of an unstructured control-flow.

We conjecture that the need for more flexible PAISs, pursued by increasingly large and varying business processes, brings to a fundamental paradigm shift in modeling, especially for expressing concepts related to concurrency. The flaws described in this paper are fundamental in nature and they cannot be overcame by simply adding new language constructs to existing unstructured BPMLs, while adding restrictions to tokens or adopting a more structured control-flow simply distort the peculiarities of this kind of languages. We hope that the presented analysis can help the design of the next generation of BPMLs.

Acknowledgments. We are grateful to Marcello La Rosa for his comments and suggestions that helped us to improve the quality of the original manuscript.

References

1. Dumas, M., van der Aalst, W.M., ter Hofstede, A.H.: Process-Aware Information Systems: Bridging People and Software Through Process Technology. Wiley-Interscience, Hoboken (2005)
2. Bandara, W., Indulska, M., Chong, S., Sadiq, S.: Major issues in business process management: an expert perspective. BPTrends (October 2007)

3. Mutschler, B., Reichert, M., Bumiller, J.: Unleashing the effectiveness of process-oriented information systems: Problem analysis, critical success factors, and implications. IEEE Transactions on Systems, Man, and Cybernetics, Part C 38(3), 280–291 (2008)
4. Adams, M., ter Hofstede, A.H.M., van der Aalst, W.M.P., Edmond, D.: Dynamic, extensible and context-aware exception handling for workflows. In: OTM Conferences (1), pp. 95–112 (2007)
5. Weber, B., Reichert, M., Rinderle-Ma, S.: Change patterns and change support features - enhancing flexibility in process-aware information systems. Data Knowl. Eng. 66(3), 438–466 (2008)
6. Sadiq, S.W., Orlowska, M.E., Sadiq, W.: Specification and validation of process constraints for flexible workflows. Inf. Syst. 30(5), 349–378 (2005)
7. Müller, D., Reichert, M., Herbst, J.: A new paradigm for the enactment and dynamic adaptation of data-driven process structures. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 48–63. Springer, Heidelberg (2008)
8. Pesic, M., van der Aalst, W.M.P.: A declarative approach for flexible business processes management. In: Business Process Management Workshops, pp. 169–180 (2006)
9. Reijers, H.A., Mendling, J.: Modularity in process models: Review and effects. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 20–35. Springer, Heidelberg (2008)
10. von Mayrhauser, A., Marie Vans, A.: Program comprehension during software maintenance and evolution. Computer 28(8), 44–55 (1995)
11. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer-Verlag New York, Inc., Secaucus (2007)
12. Russell, N., Ter Hofstede, A.H.M., Mulyar, N.: Workflow control-flow patterns: A revised view. BPM center report BPM-06-22, bpmcenter.org. Technical report (2006)
13. Russell, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Workflow data patterns: Identification, representation and tool support. Conceptual Modeling - ER, 353–368 (2005)
14. Sadiq, S., Orlowska, M., Sadiq, W., Foulger, C.: Data flow and validation in workflow modelling. In: ADC 2004: Proceedings of the 15th Australasian database conference, Darlinghurst, Australia, pp. 207–214. Australian Computer Society, Inc., Australia (2004)
15. Lee, E.A.: The problem with threads. Computer 39(5), 33–42 (2006)
16. Zapf, M., Lindheimer, U., Heinzl, A.: The myth of accelerating business processes through parallel job designs. Inf. Syst. E-Business Management 5(2), 117–137 (2007)
17. Ouyang, C., Dumas, M., Breutel, S., ter Hofstede, A.H.M.: Translating standard process models to BPEL. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 417–432. Springer, Heidelberg (2006)
18. van der Aalst, W.M.P., Aldred, L., Dumas, M., ter Hofstede, A.H.M.: Design and implementation of the YAWL system. In: Persson, A., Stirna, J. (eds.) CAiSE 2004. LNCS, vol. 3084, pp. 142–159. Springer, Heidelberg (2004)
19. Object Management Group (OMG). Business Process Modeling Notation (BPMN) version 1.1. (January 2008)
20. JBoss Enterprise Middleware Red Hat. JBoss jBPM (2008)
21. OMG. Unified modeling language: Superstructure, version 2.1.1. Technical Report formal/2007-02-03, Object Management Group (2007)

22. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Inf. Softw. Technol. 50(12), 1281–1294 (2008)
23. Kiepuszewski, B., ter Hofstede, A.H.M., Bussler, C.J.: On structured workflow modelling. In: Wangler, B., Bergman, L.D. (eds.) CAiSE 2000. LNCS, vol. 1789, pp. 431–445. Springer, Heidelberg (2000)
24. Liu, R., Kumar, A.: An analysis and taxonomy of unstructured workflows. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005. LNCS, vol. 3649, pp. 268–284. Springer, Heidelberg (2005)
25. Netzer, R.H.B., Miller, B.P.: On the complexity of event ordering for shared-memory parallel program executions. In: Proceedings of the 1990 International Conference on Parallel Processing, pp. 93–97 (1990)