

Representing Business Processes through a Temporal Data-Centric Workflow Modeling Language: an Application to the Management of Clinical Pathways

Carlo Combi, *Member, IEEE*, Mauro Gambini, Sara Migliorini and Roberto Posenato

Abstract—Workflow technology has emerged as one of the leading technologies in modeling, redesigning, and executing business processes in several different application domains. Among them, the representation and management of health and clinical processes have been obtaining a growing interest. Such processes are in general related to the way each health organization provides the required health care services: health and clinical processes underlie the specification and application of clinical protocols, clinical guidelines, clinical pathways, and the most common clinical/administrative procedures.

Current workflow systems are lacking in an effective management of three general key aspects that are common (not only) in the clinical/health context: data dependencies, exception handling, and temporal constraints. For example, a laparoscopic intervention could need the results of the concurrent bioptic analysis to be properly concluded, while exceptional recovery activities have to be performed in case of emergency evidence during the standard treatment; on the other hand, the successful application of a fibrinolytic therapy requires a maximum delay of 30 minutes after the emergency department admission.

In this paper, we propose *TNest*, a new advanced, structured and highly modular workflow modeling language that allows one to easily express data dependencies and time constraints during process design, besides to exception handling and compensation activities. As for temporal constraints, we focus here on *temporal controllability*, which is the capability of executing a workflow for all possible durations of all tasks satisfying all temporal constraints. Moreover, we analyze the computational complexity of the temporal controllability problem in *TNest* and we propose a general algorithm to check the controllability. All the features of *TNest* have been considered to model clinical pathways from classical clinical guidelines, i.e., those for the management of STEMI patients, published by the American College of Cardiology/American Heart Association, we will use throughout the paper as a motivating scenario.

Index Terms—Healthcare systems, information systems, workflow management, time management.

I. INTRODUCTION

Workflow technology is one of the leading technologies for the management of business processes in several different application domains. A Workflow Management System

Carlo Combi and Roberto Posenato are with the Department of Computer Science, University of Verona, 37134 Verona, Italy (email: {carlo.combi—roberto.posenato}@univr.it).

Sara Migliorini and Mauro Gambini are with the Department of Public Health and Community Medicine, University of Verona, 37134 Verona, Italy (email: sara.migliorini@univr.it).

(*WfMS*) has to coordinate the execution of *cases*, i.e., business processes, taking care of resource allocation and procedural constraints. Each case is an instance of a *schema*, which may correspond to several cases executed in different (possibly contemporary) moments: a workflow schema (*process model*) is a formal description of an organizational process where each atomic unit of work (*task*) is assigned to one or more processing entities (*agents*). An agent may be a software application (e.g., an electronic medical record system), a human (e.g., a physician) or a combination of both (e.g., a physician using a medical decision support tool).

Workflow technology has been adopted in several application domains: Aerospace/Defense, Heavy/Light manufacturing, Chemicals/Energy, Computers/Consumer Electronics/Software, Education, Financial Services/Insurance, Food/Beverage, Government/Military, Healthcare/Medical Equipment, Leisure/Entertainment/Travel, Professional/Business Services/Consulting, Retail and Wholesale, Telecommunications, Utilities [1]. Among the different application domains, medical organizations use *WfMSs* to streamline, automate, and manage medical processes that depend on clinical information systems and human resources: supporting clinical guidelines, managing admission-discharge-transfer processes, and supporting clinical pathways [2], [3], [4]. A clinical pathway represents a set of possible schedules of medical and nursing procedures, including diagnostic tests, medications, and consultations designed to effect an efficient, coordinated and structured program of treatment. According to this general meaning, a clinical pathway may result either from a process-oriented view of most health and clinical procedures (often involving different organization units of a health care institution) or from a concrete application, adaptation, and execution of a clinical guideline within a specific healthcare organization [5].

Without loss of generality, in the following we will mainly consider the healthcare domain: indeed, healthcare process modeling and management arise several general issues common to other application domains. It is worth noting that in the healthcare domain these issues become crucial to be solved, to guarantee the right applicability of *WfMSs*. In particular, we will focus on three key aspects of workflow technology that need to be properly addressed to make *WfMSs* usable in real world (health) scenarios: *data*

dependencies, exception handling, and temporal constraints.

Data dependencies arise when different tasks are intertwined with respect to data they need to be properly executed; for instance, a laparoscopic intervention could need the results of the concurrent bioptic analysis to be properly concluded: in this case data synchronization among concurrent tasks has to be explicitly represented. A related issue is that, since schemata of workflows are often huge and with complex connections among tasks, it is important to prevent the possibility of having deadlocks and/or lacks of synchronization: a structured modeling language can guarantee this and it has been recently shown that a structured modeling language, with proper constructs to manage data dependencies, can be defined without reducing its expressiveness w.r.t. unstructured workflow languages [6]. Roughly speaking, a schema is structured if it is formed by subgraphs with a single-entry and a single-exit point and the type of its routing constructs is properly matched.

Exception handling can be used to model unlikely situations that rarely occur and deviate from standard procedures, requiring the execution of special recovering activities. For instance, an acute rupture of the intraventricular spectrum (VSR) is estimated to occur in fewer than 1% of patient with ST-segment Elevation Myocardial Infarction (*STEMI*), and it requires a urgent cardiac surgical repair.

On the other side, temporal constraints are always present and their management is mandatory to successfully complete a workflow: for example, to successfully apply the fibrinolytic therapy to patients with *STEMI* a maximum delay of 30 minutes must be considered with respect to the emergency department admission [7]. An important concept related to temporal constraints is that of *controllability* [8]: controllability is the capability of executing a workflow for all possible durations of all tasks and satisfying all temporal constraints. This is of paramount importance in the clinical domain because task durations are contingent (e.g., it is possible to set up a duration range for a therapy but the *WfMS* is aware of the effective duration only after the therapy conclusion).

Available *WfMS* and research prototypes currently offer a limited support to all these aspects [9]. According to this scenario, in this paper we will deal with the following original aspects:

- 1) We propose a new structured, data-centric workflow modeling language called *TNest* that allows one to express data dependencies, exception handling and time constraints in business processes. constructs and advanced temporal validation methods.
- 2) We propose a new formal definition of execution strategies and controllabilities of *TNest* schemata.
- 3) We propose a new general algorithm to determine the optimal temporal durations for a controllable *TNest* workflow schema; then, we discuss the computational complexity of checking different kinds of controllability.
- 4) We show the specification through *TNest* of the process related to the management of *STEMI* patients, according to the guidelines published by the

American College of Cardiology/American Heart Association [7].

- 5) We discuss some architectural aspects related to the integration of *TNest* in process-aware clinical information systems.

With respect to the preliminary work [10], several topics are completely new to this paper: exception handling, the formal definition of execution strategies and controllability, and a new approach to check the controllability of *TNest* schemata based on the well-known formalism named Simple Temporal Network with Uncertainty (*STNU*) [11]. Moreover, the clinical domain has been studied in more detail considering new clinical features mainly related to exception handling and temporal constraints. Since in clinical domains some widely adopted clinical information systems exist, we discuss also a possible integration of *TNest* model in such platforms.

In more details, in Section II we highlight the difference between structured versus unstructured workflow models, then we introduce the relevant contributions towards the controllability, and, finally, we describe some relevant research directions in the area of computer-based support to clinical processes, guidelines, and pathways. Section III presents the *STEMI* guideline, we use as motivating scenario throughout the paper, and the preliminary modeling by *YAWL* workflow system. In Section IV we present, in a detailed way, *TNest* control-flow, data-flow, and temporal constructs. In Section V, we propose a new formalization of execution strategies and controllabilities for *TNest* schemata. In Section VI, we discuss how to check the controllability for the main *TNest* constructs and, then, we propose a general algorithm in order to determine the kind of controllability, if any, of an input *TNest* schema and, if the schema is controllable, the optimal durations for the workflow components. In Section VII *TNest* schemata for the overall *STEMI* process and for the treatment and for the drug therapy subprocesses are proposed and discussed showing the suitability of *TNest* for modeling such clinical process. In Section VIII, we discuss the role of *TNest* in a modern *WfMS* architecture as well as in a wider setting of process-aware clinical information systems. In Section IX, we summarize the main results of this work and sketch some future work.

II. RELATED WORK

Traditional modeling languages, like *YAWL* and *BPMN*, are classified as *unstructured* because they allow free composition of constructs without worrying much about type and position of the connected elements. However, a modeling approach that promotes the use of structured compositions whenever possible offers several advantages. First of all, different research efforts confirm that it is a good practice to adopt a structured control-flow design because structured models are more comprehensible, modular and generally contain less errors than their unstructured counterparts [12], [13]. Unfortunately, traditional modeling languages cannot enforce a structured composition without losing

expressiveness: it can be shown that there exist well-behaved unstructured models that cannot be transformed in fully structured ones in any obvious way [14]. To obviate this problem, [6] introduces a novel modeling language, called *NestFlow*, able to enforce a fully structured control-flow design without compromising expressiveness.

Regarding temporal constraints and the concept of controllability of workflow schemata, two main research directions may be relevant: i) the refinement of methodologies for business-process modeling and the extension of *WfMSs* in order to consider different kinds of temporalities; ii) in the AI-related area of temporal constraints, the study of constraint satisfiability when some temporal constraints are not under control of the system. In [15], the authors propose a temporal conceptual workflow model that enhances the expressiveness of previous proposals in representing temporal constraints, such as those related to tasks and connectors. One innovative aspect of the proposal is the standardization in expressing temporal constraints among tasks or connectors; temporal constraints are expressed by ranges representing lower and upper bound of the constraints. Furthermore, the authors propose different kinds of temporal constraints consistency and exhibit some algorithms to check the consistency of a workflow schema both at design time and at run-time. In [16], the authors propose ten time patterns used in process-aware information systems and provide a systematic literature review of existing systems considering temporal issues in workflows.

In AI area, Vidal et al. propose an extension of the Simple Temporal Network, the Simple Temporal Network with Uncertainty (*STNU*), where edges, i.e., constraints, are divided into two classes: *contingent links* and *requirement links* to manage the likely uncertainty about the duration of processes. Indeed, contingent links represent processes of uncertain duration, where finish time points (i.e., *STNU* nodes) are decided by Nature within the limits imposed by the bounds defined on the contingent links. Requirement links represent all the other processes whose finish time points are controlled by the agents that execute processes. Informally, *controllability* refers to the capability of specifying all the time points controlled by agents, satisfying all the requirement and contingent links. In particular, *dynamic controllability* ensures that it is possible to specify at run-time the time points controlled by agents only by knowing the duration of the already happened contingent links, without preventing any possible duration of the future contingent links [17]. Several algorithms have been proposed to check the dynamic controllability of a constraint network [18]; eventually, Morris showed that in the framework of *STNU* the checking controllability algorithm has polynomial-time ($O(n^4)$) complexity [19]. In [20], [21], the author highlights some issues in the approach proposed by Morris and Muscettola and proposes a stronger definition of dynamic execution strategies that fixes these problems and puts the checking algorithm on a more solid theoretical foundation.

Moving to processes and temporal constraints in the healthcare motivating domain we are focusing on, the

computer-based support of clinical guidelines has been considered since the late 90's and several methodologies and systems have been proposed, as EON, GLIF, Proforma, Prodigy, GUIDE, and Asgaard-Asbru [9]. All these proposals consider the structuring of guidelines into tasks [9], [22], but adopt different approaches according to the particular modeling challenges. They deal with different aspects related to the representation of clinical guidelines: the modeling of clinical data and of the derived information, the modeling of the suitable medical knowledge, the use of standard medical terminology and message formats, and the representation of specific medical task types (as decisions, actions, and so on). In general, all these systems support, in some cases in an implicit way, the specification of sequential, parallel, and alternative executions of tasks. Even though in some proposals there is the capability of managing some kinds of temporalities both for clinical data and for the specification of clinical tasks, none of them allows the user to explicitly represent at the conceptual level temporal constraints between tasks and, at the best of our knowledge, the issue of both temporal consistency and temporal controllability of guideline specifications has not yet been addressed. Moreover, the data flow between clinical tasks is not explicitly considered during the conceptual design. Since the scope of computer supported clinical guideline systems is broader than that of topics we will deal with in this paper, let us now focus on the use of *WfMS* technology to support the design and the execution of clinical guidelines, and, more in general, of clinical processes or clinical pathways [2], [4], [23]. Several specific features have been acknowledged as relevant for the implementation of clinical workflows [2]: flexibility and uncertainty management during the execution [4], explicit representation of medical and organizational knowledge [23], sharing of clinical guidelines (i.e., processes) through different institutions [3]. Time aspects in clinical workflow modeling and management have been recently considered in [24], [25], [10].

A further research direction dealing with processes in healthcare focuses on the resource management and optimization in hospital organizations. In [26], the authors propose a hierarchical model based on Petri Nets and Unified Modeling Language (UML) to design the structure, the resources, and the dynamics of a hospital department.

In [5], the authors provide a review of implementation challenges for process-oriented health information systems by considering computerization of workflows, guidelines, and care pathways. In more details, the authors select and review 108 papers from international journals and conferences and they identify the mostly recurring challenges and research topics: among the 25 identified challenges, we mention here *Flexibility and adaptability*, *Model verification*, *Process modeling* and *Temporal abstraction* as they are related to the topics we will face in this paper.

III. THE CLINICAL SCENARIO

To motivate our proposal, let us consider as a case study the guideline for the diagnosis and treatment of *ST-segment*

Elevation Myocardial Infarction (STEMI), published by the American College of Cardiology/American Heart Association in 2004 [7]. The guideline considers the management of patients with *STEMI* symptoms from the symptom onset to the late hospital care. In particular, it recommends different kinds of treatment (i.e., Pharmacological Reperfusion, Percutaneous Coronary Intervention (*PCI*) and Acute Surgical Reperfusion) according to different patient conditions (e.g., age, early/late presentation, specific contraindications) and describes how to deal with ancillary therapies and possible complications, and so on. As preliminary step, we tried to specify the workflow schema corresponding to *STEMI* guidelines through the Yet Another Workflow Language (*YAWL*) system [27]: the specification consists of several tasks, both compound and atomic, modeling activities from the 911 call to the clinical activities. Within the clinical activities, we distinguished tasks related to pharmacological therapies, ancillary therapies, surgery, *PCIs*, monitoring and complications.

During the design of *STEMI* guidelines through *YAWL*, some limitations have been found in the representation of the following main aspects. A first limitation is related to the specification of data dependencies among tasks: in *YAWL* it is not possible to describe them at the conceptual level, even if there are some possibilities at low level by using local and global variables to exchange data among tasks. As an example, in the *STEMI* guidelines it is important to underline that surgery is allowed only after data from monitoring devices are at disposal. A second limitation is related to the specification of temporal aspects of the clinical process: in *YAWL* it is only possible to specify deadlines, while dummy tasks are needed to handle ad hoc temporal requirements. In the *STEMI* guidelines there are several temporal constraints that need to be managed as, for example, “primary *PCI* should be performed within 90 min after the patient contact (arrival at the emergency department or contact with paramedics)”, “fibrinolytic therapy must start within 30 min after the patient contact” and “after successful fibrinolysis, transfer to a *PCI* capable hospital for coronary angiography, ideally between 3 to 24 hours after start fibrinolytic therapy, may be considered”. A third issue is related to the fact that *YAWL* allows unstructured workflow schemata that could cause deadlocks or lack of synchronization [28]. For example, in *YAWL* one can specify that tasks *PCI* and fibrinolysis are alternative and then require through an AND join connector that the workflow needs the completion of both tasks to continue. Therefore, in the design of the *STEMI* guidelines it has been difficult to (manually) check that split and join constructs were suitably nested. The last limitation regards the possibility of properly manage exceptional situations that rarely occur and that deviate from the standard procedures; for instance, specific recovering activities have to be performed in case of emergency evidence during the execution of the operative choices or of the Drug Therapy sub-process.

IV. THE *TNest* MODELING LANGUAGE

Any non-trivial process has to deal with some temporal requirements and this is exceptionally true for clinical pathways where clinicians have to plan therapies, record administered drugs, track the current state of patients and so on. Workflow technology seems to be the most appropriated technology to build flexible software systems that can support human agents in carrying out their work. Unfortunately, workflow technology is not sufficiently mature for modeling and enforcing temporal constraints: indeed, temporal aspects are at best implemented with improper constructs when they are not ignored at all. The lack of a temporal support makes workflow technology less effective for end-users and developers. *TNest* is a subset of the *NestFlow* graphical Process Modeling Language (*PML*) [6] enriched with specific temporal constructs and sanity checks that allow one to express and verify time-related properties. *NestFlow* integrates a structured control-flow with an explicit representation of data-flow aspects. The name *NestFlow* has been chosen because control-flow structures are expressed by recursively nesting language constructs with a single entry and a single exit point. In this way the process design activity is less prone to errors because threads of control are confined in specific blocks and the use of a structured control-flow helps in enforcing important static properties that guarantee a safe run-time behavior. This is a critical requirement because schemata are used not only for easing human communication but also for driving software systems in order to support and coordinate complex human activities often carried out concurrently. *TNest* embraces the *NestFlow* rationale in order to provide temporal constructs and verification methods. The following subsections represent a gentle introduction to *TNest* and its main features, including exception handling are also presented here. Except for Sec. IV-E, each section focuses on a particular language aspect, as far as possible, isolated from the other ones. Sec. IV-E briefly introduces language semantics by explaining the run-time behavior of each single part and how these parts can be put together to provide a complete specification.

A. Control-Flow Aspects

This section introduces the basic graphical elements of *TNest* and its syntax. *TNest* elements are obtained from graphical primitives, like rounded boxes, diamonds, or lines, connected together in heterogeneous groups called *blocks*. In contrast with existing Process Modeling Languages (*PMLs*), each block is intended to be manipulated as a whole by a graphical editor that can also enforce the essential syntactical rules.

The *TNest* blocks are summarized in Fig. 1 using a *BNF* notation¹ that encodes the basic compositional rules. In particular, three main syntactical categories can be distinguished: the main block $\langle P \rangle$, called *Process*, non-terminal blocks $\langle B \rangle$, *Seq*, *Choice*, *Loop*, *Par*, *Catch*,

¹In the given Backus-Naur Form (*BNF*) grammar, a rule $\langle S \rangle ::= X|Y|Z$ means that the meta-symbol $\langle S \rangle$ can be replaced by one block chosen among X , Y or Z .

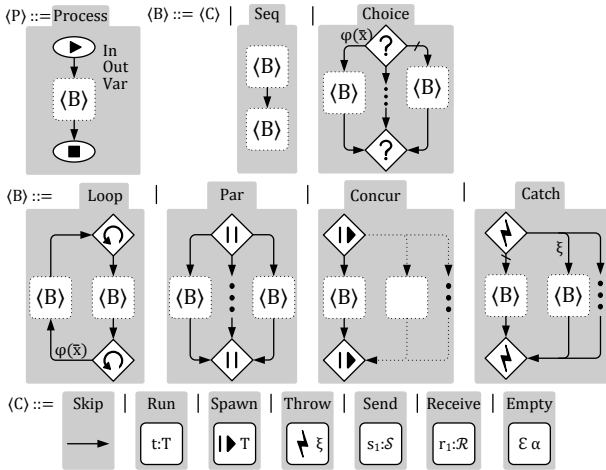


Fig. 1. The *TNest* modeling language constructs.

and *Concur*, and terminal blocks $\langle C \rangle$, *Skip*, *Run*, *Spawn*, *Throw*, *Send*, *Receive*, and *Empty*. In the following the term *block* may refer to both terminal and non-terminal blocks, while the term *command* is used as an alias for terminal blocks. Excluding the last three commands that deal with data, the other blocks are used to describe the main control-flow relations among tasks. An *edge*, depicted as a solid-line arrow, represents a control-flow between two blocks, while a *connector* denotes either the entry or the exit construct for *Choice*, *Loop*, *Par*, *Concur*, and *Catch* blocks. A *task* is essentially a more or less complex pre-existing process specification invoked using a *Run* command or created at run-time with a *Spawn* command. An invoked process can be a *native* procedure implemented with a general-purpose programming language or any other previously defined *TNest* process model. The ability of reusing an existing process specification is essential for supporting a uniform hierarchical decomposition that in turn enhances the overall system modularity. A native procedure can be used to implement an automatic task, drive an external program or interact with human agents by means of a graphical user interface.

A process specification, i.e., a *workflow schema*, is obtained starting from the main process $\langle P \rangle$ and by recursively nesting multiple blocks $\langle B \rangle$ and commands $\langle C \rangle$ following the grammar in Fig. 1 and some additional syntactical rules that are explained in Sec. IV-D. A sequence of blocks of arbitrary length n can be obtained by nesting $n - 1$ *Seq* blocks. *Choice* and *Par* blocks can have two or more branches. Each branch i of a *Choice* block is annotated with a condition $\varphi_i(\bar{x})$ over a set of variables \bar{x} , except the last one, which is called default branch and is marked with an oblique bar $/$. A *Par* block is used to execute in parallel two or more inner blocks that have to synchronize at the end of the block before leaving it. A *Loop* block has two branches: both of them can be expanded with further inner blocks or they can be also used alone closing the other branch with a *Skip* command. *Concur* is a block with initially two branches, one depicted as a solid line that represents the main flow, and one with a shaded line that

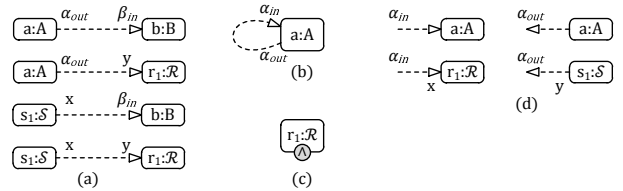


Fig. 2. Different combinations of *TNest* links. Tasks are denoted by A and B and the corresponding task instance identifiers by lower-case letters. Streams are denoted by the initial Greek letters α and β , while x and y are variables. S and R denote commands *Send* and *Receive*, respectively.

denotes the possibility of adding one more branches at run-time with a *Spawn* command, as explained in Sec. IV-E. *TNest* is designed not only for creating schemata but also for displaying process executions: block *Concur* offers a basic mechanism to make visible the creation and destruction of dynamic entities by growing and shrinking the displayed model. *PMLs* usually do not offer any representation of this dynamic behavior and concurrent instances are usually left implicit. *Catch* is a block with two or more branches. The first branch, depicted as a solid line, represents the main flow; the other branches, each annotated with a type ξ_i , represent the alternative execution of the block if an exception of type ξ_i occurs on the first branch.

The formal semantics of the main *TNest* constructs is given in Appendix A.

B. Data-Flow Aspects

A workflow schema shall be accompanied with the declaration of input and output streams as well as local variables using the keywords *in*, *out* and *var*, respectively. Input and output streams represent the interface of the process, while variables capture the main part of its internal state.

It is assumed that every stream, variable and process has a type and an identifier, for instance $x : \text{Int}$ denotes a variable with identifier x and type *Int*. The same notation holds for streams, but for convenience a subscript *in* or *out* is added to the identifier, especially when no declaration is given. A process instance invoked in a *Run* command is declared in a similar way with the difference that its identifier can be omitted in the graphical representation.

A *stream* is simply a queue for objects of the same type that can be used for modeling, not only the information flow, but also the set of objects needed and produced by a task. An instance of A can refer to one of its own interface stream α through the dot notation $\text{this}.\alpha$, where *this* is a language keyword. Similarly, a stream α of an internal component instance $b : B$ is referred by $b.\alpha$, where b is the instance identifier and α is an interface stream of B . The dot-notation ensures that all streams in a component are uniquely identified and the keyword *this* can be left implicit.

The flow of objects among internal tasks of a schema is expressed through the notion of *link* that is a unidirectional connection between two streams. A link is graphically denoted using a dashed line with a hollow arrow pointing

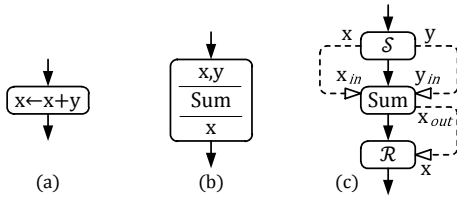


Fig. 3. (a) Example of an expression involving variables. (b) Functional-style parameter passing. (c) The two notations can be implemented with message passing.

to the task that exposes the input stream. Depending on involved commands, link ends are also annotated with stream or variable identifiers as summarized in Fig. 2.

Links can be distinguished in internal and external ones, as in Fig. 2(a)-(b) and Fig. 2(d), respectively. Internal links connect two streams inside the same process, external links are dangling dashed lines that represent an interaction with the environment.

One strength of *TNest* resides on the possibility to hide links and their related commands any time they can be subsumed by the main control-flow. For instance, internal links with the same source and target can be grouped into a unique *collapsed link* and subsumed by a control-flow relation with the same direction; moreover, *Receive* and *Send* commands with hidden links can be subsumed as well. The remaining links mostly describe interactions among concurrent tasks. Message passing is sufficiently expressive to encode parameter passing mechanism, hence parameter passing notation can be considered as a syntactic abbreviation for representing message exchange.

A process can declare zero or more variables with their own type. Variables are visible only inside the component where they have been declared and their scope does not extend to components contained in it. For shortening the presentation, *TNest* does not include operators to express computations on variables: it can be assumed without losing expressiveness that any computation can be implemented as a native task eventually annotated with the corresponding expression. For example, a task denoted with $x \leftarrow x + y$, as in Fig. 3, means that exists a native task with two input streams $x_{in} : \text{Int}$, $y_{in} : \text{Int}$ and an output stream $x_{out} : \text{Int}$ that taken the value of the variables x and y in the same scope computes their sum and stores it again in x . The connection between variables and streams is obtained by means of *Send/Receive* commands and links. Thanks to this kind of encoding, the language gains compactness maintaining the same small set of primitives.

C. Specification of Temporal Constraints

As already introduced in previous sections, in this paper we want to propose a data-driven workflow language where many temporal aspects can be specified and managed. In *TNest* there are two main kinds of temporal constraints: *activity duration* and *relative constraint*.

An activity duration represents the allowed temporal spans for the execution of a either a task, a connector,

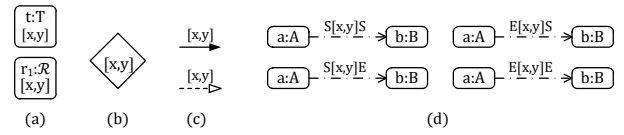


Fig. 4. *TNest* constructs with temporal constraints. (a) Duration constraint associated to a task, or a *Receive* command, respectively. (b) Duration constraint associated to a connector, notice that the connector is empty to be general. (c) Delay defined on a control-flow relation, or a data-flow link, respectively. (d) Relative constraint between the two tasks.

or an edge (see Fig. 4(a), (b) and (c), respectively) and it is assumed that each activity, but *Spawn*, *Send* and *Throw*, has a duration specification set by the designer. Activity durations are expressed by ranges like $[\text{MinD}, \text{MaxD}]$ *Granularity* where $0 \leq \text{MinD} \leq \text{MaxD} \leq \infty$ and *Granularity* stands for the time unit used. If a designer does not set a duration, it is assumed to be $[1, \infty]$ *MinGranularity*, where *MinGranularity* is the minimum time granularity used by the *WFMS* managing the workflow schemata (0 *MinGranularity* is not allowed because it is not possible to execute an activity without consuming time). Since *Spawn*, *Send*, and *Throw* are non-blocking activities and are used as milestone by the run-time engine to start other blocks or messages, we assume that they have a fixed unit duration that cannot be modified by the designer. The model allows also the setting of a duration for edges: it may 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. The duration associated to a link represents the allowed *delivery time* of a message once it is generated. In general, duration constraints can be modified during the controllability check at design time (more details in the following section) or at run-time in order to satisfy all the given constraints. Nevertheless, such assumption is not suitable to represent real environments where tasks are executed by external agents usually requiring a non negotiable time to complete an assignment. Therefore, we assume that the duration constraint for tasks is a non modifiable constraint and neither controllability check nor run-time engine can modify it.

The other kind of temporal constraints is *relative constraints*. They allow the expression of several other kinds of temporal constraints among activities. A relative constraint limits the time distance (duration) between the starting/ending instants of two non-consecutive workflow blocks. It is graphically represented as a dash-dot-dash edge between the two blocks (see Fig. 4(d)). The label of the edge specifies the constraint according to the following pattern: $\langle I_F \rangle [\text{MinD}, \text{MaxD}] \langle I_S \rangle$ *Granularity*, where i) $\langle I_F \rangle$ marks which instant of the First activity to use ($\langle I_F \rangle = S_{\langle \text{activity} \rangle} \mid E_{\langle \text{activity} \rangle}$ as the starting/ending execution instant, respectively; the subscript can be omitted if it is clear from the context); ii) $\langle I_S \rangle$ marks the instant for the Second activity in the same way; iii) $[\text{MinD}, \text{MaxD}]$ *Granularity* represents the allowed range for the time distance between the two instants $\langle I_F \rangle$ and $\langle I_S \rangle$. We assume that $-\infty \leq \text{MinD} \leq \text{MaxD} \leq \infty$. 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 I_S : if the global time spent to execute all activities between I_F and I_S is less than MinD , then the *WfMS* has to dynamically manage a suitable action (like to sleep, for example) that depends from the specific applications. A finite negative MaxD value expresses that the I_S has to occur at least $|\text{MaxD}|$ instants before I_F . In general, if a designer does not specify the granularity of a range, it is assumed that the granularity is MinGranularity .

D. Well-Formed Schemata

In general, a schema is *well-formed* if it can be executed without unexpected exceptions. A well-formed schema satisfies some properties that can be statically checked without executing it. These properties are important because they can guarantee the presence of a good run-time behavior that can be hard to prove otherwise. As an example of such properties, a link can connect only an output stream with an input stream of the same type or a super-type; moreover, a `Spawn` command can be placed only inside a `Concur` block; and so on. An exhaustive formalization of all *TNest* well-formedness properties is out of the scope of this paper; instead this section wants to introduce the most relevant ones, namely i) *only one place per component instance* and ii) *no shared variables among parallel branches*. The first rule guarantees that there are no concurrent executions of the same component which may leave the process in an inconsistent state. At the same time it simplifies data-flow graphical specification because the source and the target of a link are uniquely identified without explicitly stating task identifiers. The second rule prunes away non-deterministic executions caused by the exact timing of events that are not completely under the designer's control.

Both properties can be relaxed at the expense of making their check more complex. For example, two or more parallel branches can access in read-only mode to the same variable without causing inconsistencies; read-only access can be easily verified by checking whenever a variable appears as an output or a left-value in an assignment statement.

Since the relative constraint concept is quite general and some block constructs have a complex behavior, some unfitting settings are possible. Hence, we propose the following five general construction rules.

- 1) Relative constraints cannot be set for 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. 5(a). Further constraints between other tasks and the spawned tasks may be arbitrarily set.
- 3) Inside a loop, it is possible to set a relative constraint between two tasks but on different cycles of the loop using a specific notation and with some limitations.

For example, in Fig. 5(a) the relative constraint between `B` and `D` has label $\blacktriangleright S[x, y]S$ meaning that any instance of `D` has to start in $[x, y]$ time units after (i.e., $x \geq 0$) the start of the block `B` spawned before it in the same cycle iteration.

- 4) Between two connected `Send-Receive` commands there is always an implicit $S[u, \infty]E$ 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 (see Fig. 5(b), where there is an explicit constraint $S[u, q]E$), 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 `Catch` block, an implicit $E[0, 0]S$ relative constraint between the `Throw` command and the first task on the associated exception branch it is always set as shown in Fig. 5(c). Relative constraints between tasks on an exception branch and tasks outside the `Catch` block are not allowed because we assume that exceptions should hardly ever occur and allowing relative constraints between tasks on an exception branch and tasks outside it makes the controllability harder for most of the schema executions uselessly.

E. Runtime Behavior

The complete state of a process is given by the state of its streams, its variables, and the component instances contained in it. In *TNest* a component instance is *stateful*: it retains its state over multiple executions. Any new instance starts from an initial state, which may be modified by sequential executions of the component internal blocks. *TNest* blocks have the following behavior:

- `Sequence` block. It runs the first inner block $\langle B \rangle$ until completion, then it executes the next one.
- `Choice` block. It evaluates conditions $\varphi_i(\bar{x})$ associated to each branch in a fixed order. As soon as a condition is true, the corresponding branch is executed, otherwise the default (rightmost) one is chosen.
- `Loop` block. It executes blocks contained in its two branches multiple times. After the execution of the right branch in Fig. 1, condition $\varphi(\bar{x})$ is evaluated. If the condition is false the loop exits, otherwise it executes the left and the right branch in sequence. The condition $\varphi(\bar{x})$ specifies either a maximum number of iterations n or a timeout t .
- `Par` block. It executes the specified branches in parallel each with its own thread of control. The block is left only when all threads have been completed.
- `Concur` block. It is the scope of a dynamic component creation. It initially executes the main body $\langle B \rangle$ but one or more parallel branches can be added at run-time using a `Spawn`; all threads join before exiting the block.

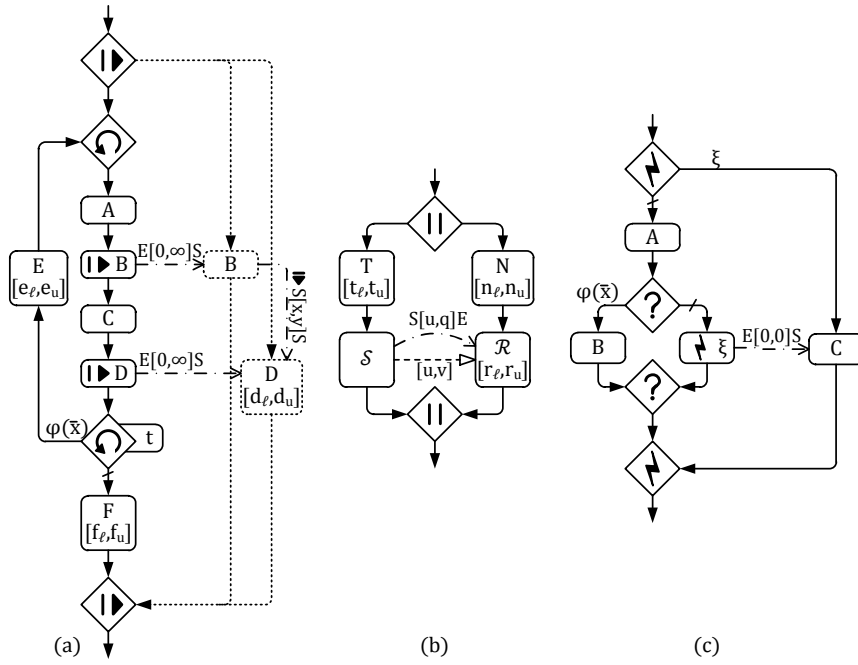


Fig. 5. (a) Specification of temporal constraints in presence of a spawn command. (b) Specification of temporal constraints between a Send and a Receive command. (c) Specification of temporal constraints in presence of a Catch block.

– **Catch block.** It executes the default branch and if an exception of type ξ_i is raised inside it, the current execution is interrupted and resumed from the branch annotated with the proper exception type ξ_i to handle the exceptional situation. For example, in Fig. 1 the non-terminal block $\langle B \rangle$ in the second branch is executed when an exception of type ξ is raised by a Throw command in the default branch. An exception raised inside a parallel branch that does not contain a corresponding Catch block, causes also the interruption of all the other tasks running in the parallel branches, through the raise of an interrupt exception on those branches, in order to revert the entire block. All these different exceptions (i.e., the initial exception and the various interrupt ones) are grouped into a single exception before leaving the parallel block.

– **Skip.** It is useful for obtaining specific control-flow structures from generic ones; for instance, the usual *while* and *repeat-until* loops can be obtained replacing the right or the left branch of a Loop with it, respectively.

– **Run.** It executes a component instance and the current thread of control is suspended until the component completes. The command has no special symbol, it is only labeled with the invoked component type and optionally with the component instance identifier.

– **Throw.** It raises an exception of the specified type, reverting recursively all blocks that contain it until a proper handler is reached.

– **Spawn.** It creates a new task instance $t : T$ that is immediately executed into a new parallel branch added to the inner Concur block containing the command.

– **Send.** It inserts the value of one or more variables into one or more corresponding output streams. A Send is non-blocking: the execution continues with the next block

without waiting.

– **Receive.** It 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 expires. A multiple Receive stores the first arrived object from a stream α_{in}^i into the corresponding variable x_i , and continues the execution; this behavior is called *or-receive*. A sequence of Receive commands can be grouped into a unique *and-receive* which waits for an object from each connected stream before proceeding and is denoted as in Fig. 2(c).

– **Empty.** It removes all objects contained in a specified stream α . An Empty command does not have its own input or output streams, but it is annotated with the input stream of an existing Receive command contained in the workflow. The Empty is non-blocking: if the specified stream a is already empty, it does not wait. An Empty command can be seen as a Receive inside a Loop which is iteratively performed until all objects in its input stream α have been consumed. However, this command cannot be really simulated with existing constructs, because an Empty consumes the object previously enqueued in the input stream of another Receive placed somewhere of the workflow, and two commands cannot share the same streams. Since streams hold the enqueued objects until they have been consumed, the Empty command is useful, for instance, for resetting the state of a Receive placed inside a Loop by consuming all remaining objects before a new iteration is performed, assuming that each iteration has to be performed on its own data.

A Send has only output streams and a Receive only input ones: regardless of its name, a stream may have a different direction depending on the internal or external

perspective. Furthermore, `Send` and `Receive` commands can be viewed as special component instances with their own identifiers; hence, each variable x involved in a `Send` $s:\mathcal{S}$ can be considered as an output stream $s.x_{out}$, while each variable y of a `Receive` $r:\mathcal{R}$ can be considered as an input stream $r.y_{in}$.

V. CONTROLLABILITY IN *TNest*

Here, we extend the approach proposed in [8], [29] to *TNest* and propose a completely new formalization of the controllability of workflow schemata. In the first subsection we propose some preliminary definitions and the formalization of the concept of execution of a schema (*schedule*); moreover, we propose a characterization of possible *execution strategies* with respect to their applicability to real systems. In the following subsection, we describe two possible kinds of controllability of schemata based on the possible execution strategies for their *wf-paths*.

A. Formalizing Execution Strategies

After the specification of the runtime behavior of *TNest* components, to formally define the concepts of execution strategies and controllability of a *TNest* schema, we need to introduce some preliminary concepts.

If a *TNest* schema contains one or more `Choice` or `Loop` or `Catch` blocks, not all the its cases perform exactly the same set of blocks.

Regarding the presence of a `Loop` block in a *TNest* schema, it is possible to show that an equivalent schema is possible without using the `Loop` construct.

Lemma 1: Each `Loop` block, even containing one or more `Spawn` commands, can be represented by an appropriate sequence of copies of the block each controlled by a `Choice` block and `Skip` command preserving all temporal constraints.

Proof: Since it is always possible to determine in advance an upper bound to the maximum number of *iterations* of a `Loop`, a workflow containing a `Loop` can be rewritten as a loop-free one. For example, a do-while cycle subgraph can be replaced by a subgraph containing *iterations* copies of the original loop body. In particular, the first copy is followed by (*iterations*-1) nested `Choice` blocks each one containing two branches with a condition on the first one equals to the loop condition. The first branch is populated with a copy of the loop body followed by the subsequence `Choice`, while the second branch contains a `Skip` command.

If a cycle contains a `Spawn` block, the described unrolling phase has also to set up the necessary parallel branches in order to properly connect the possible relative constraints between blocks within the cycle and dynamically generated tasks as exemplified in Fig. 6. In more details, the unrolling phase creates as many parallel branches as the number of cycles, each of them as copy of the dotted parallel branch of the schema and in one-to-one correspondence with a cycle subgraph. Possible relative constraints depicted in the

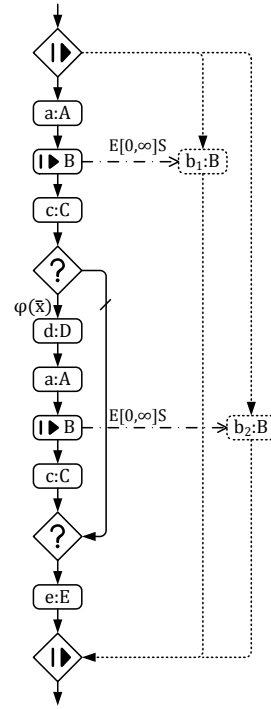


Fig. 6. Unrolling of a loop of two cycles containing a `Spawn`. This transformation requires to relax the first property of well-formedness, i.e., *only one place per component instance*, because the instance identifiers a and c appears multiple times in the same process: one time for each cycle. Anyway, this relaxation is admissible because they run in sequence and no concurrent execution of the same instance are possible. The use of the same task instance is necessary because tasks are stateful; therefore, the execution of the same instance multiple times in sequence can produce different effects from the sequential execution of different instances of the same task.

schema between blocks within a cycle and dynamically generated blocks are copied in the cycle-free graph considering each cycle subgraph and the corresponding parallel branch. ■

Regarding `Catch` blocks, we observe the following fact.

Fact 1: Each `Catch` block containing one or more `Throw` commands can be viewed as a set of alternative cases, one corresponding to the execution without any exception, and the other ones corresponding to each possible execution of a `Throw` command.

In general, in a `Catch` block there can be different `Throw` commands in different positions, even in branches associated to exceptions (nested exceptions). Here, without loss of generality, we consider only cases where one or more `Throw` commands are present into the default branch. For example, let us consider Fig. 7(a). There are two possible cases:

- 1) the first one occurs when no exception is raised, as depicted in Fig. 7(b).
- 2) the second one occurs when the exception ξ is raised and, therefore, tasks G and H are executed just after the `Throw` command $\downarrow \xi$. It is worth observing that all activities after `Throw` command are skipped but, in order to maintain a right graph structure, only tasks are removed from the graph while the connector

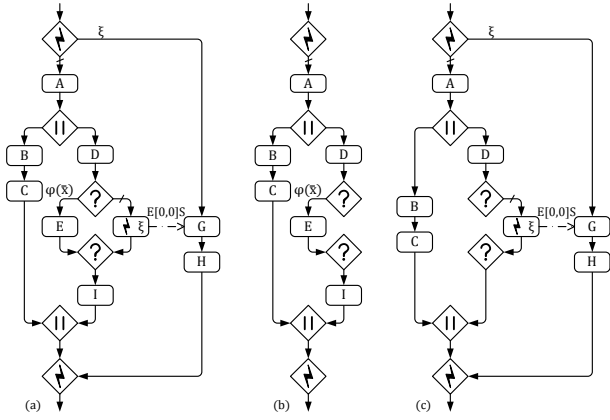


Fig. 7. Controllability analysis of a `Catch` block. In (a) a simple `Catch` block is depicted. In (b) the corresponding default branch. In (c) the branch of the ξ exception.

are maintained with duration $[0, 0]$ as depicted in Fig. 7(c). If there are parallel branches to the branch containing the `Throw` command, such branches are part of the case, even if it is possible that not all tasks of these branches are really executed according to the semantics of exception raising. Such case containing all parallel branches represents the “worst” possible execution where all possible constraints must be considered.

Now, we are ready to introduce the concept of *wf-path* that allows the identification of which blocks are performed during a case.

Definition 1 (wf-path): Given a *TNest* schema where each possible `Loop` block has been replaced according to Lemma 1, a workflow path (*wf-path*), usually denoted by p , denotes a connected maximal workflow subgraph of the schema containing the starting node, $\langle \blacktriangleright \rangle$, and ending one, $\langle \blacksquare \rangle$, and such that all `Choice` connectors have exactly one branch and each `Catch` block contains either the default branch without any `Throw` command or the default branch with exactly one `Throw` command and the corresponding exception branch as described in Fact 1.

A *wf-path* p can be also briefly described by a string containing the task identifiers of the *wf-path* sorted w.r.t. their execution order and separated by a dash if the order is sequential or by a double vertical bar if the order is parallel, as illustrated in the following example. If there exists a `Throw` command, it is explicitly represented by \downarrow followed by the exception type and the representation of the corresponding exception branch.

Example 1 (wf-path): Consider the schema in Fig. 8, the *wf-path* A-B-C represents a *wf-path* where tasks are executed in the sequential order A, B, C while D-(E—F) represents a *wf-path* where task D is executed before concurrent tasks E and F.

Note that the set WfP_{TN} of all *wf-paths* of a *TNest* schema TN may have an exponential cardinality w.r.t. the number of `Choice` blocks present in the schema.

Durations of tasks cannot be decided by a *WfMS* executing a schema case but they can only be considered

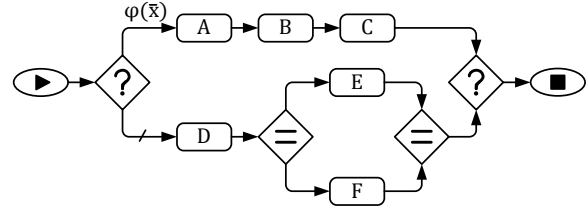


Fig. 8. A simple workflow schema to show two alternative *wf-paths*: A-B-C and D-(E—F)

to determine the durations of other connectors/delays. A *situation* represents one possible combination of task durations in an execution.

Definition 2 (situation): Let \mathcal{S} be a sub schema containing k tasks, T_1, T_2, \dots, T_k , with respective duration ranges, $[x_1, y_1], \dots, [x_k, y_k]$. $\Omega_{\mathcal{S}} = [x_1, y_1] \times \dots \times [x_k, y_k]$ is called the *space of situations* for \mathcal{S} and any $w = (d_1, \dots, d_k) \in \Omega_{\mathcal{S}}$ is called *situation*. We may write Ω instead of $\Omega_{\mathcal{S}}$ when the set of tasks is clear from the context.

The pair (p, ω) , where p is a *wf-path* and ω is a situation for tasks in p is called *wf-path situation*:

Definition 3 (wf-path situation): Given a *TNest* schema TN , a *wf-path situation* is any pair (p, ω) , where $p \in WfP_{TN}$, and $\omega \in \Omega_p$ is a situation defined considering tasks of p . The set of all *wf-path situations* (for TN) is contained in $WfP_{TN} \times \Omega_{TN}$.

In a *wf-path situation* all task durations are known and, therefore, it is only necessary to properly select durations of connectors/delays and the start instant of the case in order to *execute the wf-path*.

More generally, the concept of *execution* is defined by specifying a *schedule* function that associates each starting/ending instant of tasks/connectors with a timestamp value.

Definition 4 (Schedule): Let $\mathcal{I}_{\mathcal{S}}$ be the set of all starting/ending instants (also called *time-points*) of tasks/connectors of a *TNest* (sub)schema \mathcal{S} . Let $\mathcal{T} \subseteq \mathcal{I}_{\mathcal{S}}$.

A *schedule* for a set of time-points \mathcal{T} is a mapping $sc : \mathcal{T} \rightarrow \mathbb{N}$ that assigns a natural number to each time-point in \mathcal{T} . The set of all schedules for any subset of \mathcal{T} is denoted by $SC_{\mathcal{T}}$.

Since every *WfMS* can manage the temporal aspects with a fixed accuracy (given by `MinGranularity`), it is possible consider the set of natural number as codomain of a schedule function without loss of generality.

A schedule defines the starting/ending instants of tasks and connectors, while an *execution strategy* allows the selection of which schedules can be used given a *wf-path situation* of a schema.

Definition 5 (Execution strategy): Let TN be a *TNest* schema. An *execution strategy* st for TN is a mapping $st : (WfP_{TN} \times \Omega_{TN}) \rightarrow SC_{\mathcal{I}_{TN}}$, such that for each *wf-path situation*, (p, ω) , returns a schedule for the set of time-points of the *wf-path* p observing the durations specified by ω .

st is called *viable* if for each *wf-path situation*, (p, ω) , the schedule $st(p, \omega)$ satisfies all the temporal constraints in p when the situation is ω . For any time-point x and *wf-path situation* (p, ω) , the execution time of x in the schedule

$st(p, \omega)$ is denoted by $[st(p, \omega)]_x$.

An example of execution strategy is a function that builds a schedule assigning as timestamp value of a time-point the first value that satisfies all constraints involving the considered time-point (*early-execution* strategy).

In general, a *TNest* case can be executed if it admits an execution strategy, but the general definition cannot be directly used in real systems because it requires to know in advance which *wf-path* situation will be considered before starting the execution. In real environments, it is required to use an execution strategy that determines a schedule in an incremental way, according to the duration of already executed tasks (partial situations) and the decisions made in the already executed choice connectors.

In order to formalize properties of such required execution strategies, we define the following preliminary concepts:

Definition 6 (Situation history): Let *TN* be a *TNest* schema, *st* be any execution strategy for *TN*, (p, ω) be any *wf-path* situation, and *t* be any natural number. Then the *history* of *t* in the situation ω , for the strategy *st* and *wf-path* *p*, denoted by $sitHst(t, p, \omega, st)$, is the subset of ω where:

- $sitHst(t, p, \omega, st) = \{d_i \mid d_i \in \omega \text{ and } d_i \text{ is associated to task } T_i \text{ and } [st(p, \omega)]_{T_{iE}} < t\}$, where T_{iE} is the ending instant of T_i .

Definition 7 (wf-path situation history): Let *TN* be a *TNest* schema, *st* be an execution strategy for *TN*, (p, ω) be some *wf-path* situation, and *t* be some natural number. Then the *history* of *t* for the *wf-path* situation (p, ω) and execution strategy *st*, denoted by $wfHst(t, p, \omega, st)$, is the pair $(\mathcal{H}_p, \mathcal{H}_\omega)$ where:

- $\mathcal{H}_p = \{x \mid x \text{ is a starting/ending instant of a task/connector in } p \text{ and } [st(p, \omega)]_x < t\}$; and
- $\mathcal{H}_\omega = sitHst(t, p, \omega, st)$.

In other words, *wf-path* situation history $wfHst(t, p, \omega, st)$ specifies the set of durations of already executed task at instant *t* and the set of all starting/ending instants of all tasks/connectors already executed at time *t*. The information of the first set could be derived from the second set, but we consider it explicitly for sake of readability.

Let $\mathcal{I}_p^- \subseteq \mathcal{I}_p$ be the subset of all starting/ending time-points of connectors and all starting time-points of tasks of a *TNest* *wf-path* *p*. \mathcal{I}_p^- contains the time-points that are not contingent and, therefore, can be assigned by a schedule according to an execution strategy. Now, considering the two concepts of history, it is possible to impose different restrictions on possible execution strategies for a schema, obtaining the following two interesting strategies:

Definition 8 (Dynamic Execution Strategy): A viable execution strategy *st* for *TN* is called *dynamic* if, given a *wf-path* *p*, for any pair of situations, $\omega_1 \in \Omega_p$ and $\omega_2 \in \Omega_p$, and any time-point $x \in \mathcal{I}_p^-$, it holds: if $[st(p, \omega_1)]_x = k$ and $wfHst(k, p, \omega_1, st) = wfHst(k, p, \omega_2, st)$, then $[st(p, \omega_2)]_x = k$.

In other words, if the execution strategy *st* in the situation ω_1 assigns the value *k* to the executable time-point *x*, then *st* must also assign *k* to *x* for any other situation ω_2 whose

wf-path situation history relative to *k* matches that of ω_1 on the same *wf-path* *p*.

Definition 9 (Fully Dynamic Execution Strategy): A viable execution strategy *st* for *TN* is called *fully dynamic* if for every pair of *wf-paths* p_1, p_2 , for every pair of situations, $\omega_1 \in \Omega_{p_1}$ and $\omega_2 \in \Omega_{p_2}$, and for any time-point $x \in \mathcal{I}_{p_1}^- \cap \mathcal{I}_{p_2}^-$ it holds: if $[st(p_1, \omega_1)]_x = k$ and $wfHst(k, p_1, \omega_1, st) = wfHst(k, p_2, \omega_2, st)$, then $[st(p_2, \omega_2)]_x = k$.

In other words, if the execution strategy *st* in the situation ω_1 assigns the value *k* to the executable time-point *x*, then *st* must also assign *k* to *x* for any other *wf-path* p_2 and situation ω_2 whose *wf-path* situation history relative to *k* matches that of ω_1 on the *wf-path* p_1 .

B. Controllability of a *TNest* Schema

Considering the two kinds of execution strategies introduced in the previous section, it is useful to classify workflow schemata with respect to the possible execution strategies they admit.

Definition 10 (Weak Controllability): A *TNest* schema is called *weakly controllable* (**WeC**) if it admits a viable dynamic execution strategy for each *wf-path*; i.e., if it is possible to perform a *wf-path* satisfying all relative constraints using allowed delays and allowed connector durations once the *wf-path* is known.

The requirement of knowing in advance which *wf-path* will be followed is a quasi-impossible condition to satisfy in real applications. Therefore, it is useful to consider more powerful execution strategies that could determine schedules given only the knowledge of the already executed activities.

Definition 11: History-Dependent Controllability A *TNest* schema is called *history-dependently controllable* (**HDC**) if it admits a fully dynamic execution strategy *st*; i.e., if it possible to perform any *wf-path* satisfying all relative constraints using allowed delays and allowed connector durations knowing only the durations of connectors/tasks already executed.

In this case the duration range of a connector/edge could depend on the executed tasks (i.e., the history), but does not prevent the *WfMS* to execute any possible future activity. According to such dependency, a given connector/edge may have different and disjoint duration ranges, each one corresponding to a different history².

In general, we will say that a *TNest* schema is *controllable* if it is *History-Dependently Controllable*.

VI. CHECKING THE CONTROLLABILITY OF A *TNest* SCHEMA

In this section we first discuss how to check the controllability of a schema showing how to map a single *wf-path* to a *Simple Temporal Network with Uncertainty* (*STNU*) and how to exploit the well known results about *STNU* controllability. Then, we show how to check the controllability in general schemata by providing a new checking algorithm.

²As a subcase of **HDC**, in [30] we defined a schema as *Strongly Controllable* if it is **HDC** and each connector/edge has only one duration range for all possible histories.

Let us now start recalling the basic concepts and results about *STNUs*.

A. *STNU*: definition and main properties

Following [11], a *Simple Temporal Network with Uncertainty (STNU)* is a set of *time-point variables* (hereinafter, time-points) together with *temporal ordinary constraints* and *contingent links*. Each ordinary constraint has the form $Y - X \leq \delta$, where X and Y are time-points and δ is a real number. Each contingent link has the form, (A, u, v, C) , where A and C are time-points and $0 < u < v < \infty$. A is called the *activation time-point*; C is the *contingent time-point*. Once A is executed, C is guaranteed to execute such that $C - A \in [u, v]$. However, the particular time at which C executes is uncontrollable. Instead, it is only observed as it happens.

More formally, let $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L})$ be an *STNU*, where \mathcal{T} is a set of time-points, \mathcal{C} is a set of constraints, and \mathcal{L} is a set of contingent links. The graph associated with \mathcal{S} has the form, $(\mathcal{T}, \mathcal{E}, \mathcal{E}_\ell, \mathcal{E}_u)$, where each time-point in \mathcal{T} serves as a node in the graph; \mathcal{E} is a set of *ordinary edges*; \mathcal{E}_ℓ is a set of *lower-case edges*; and \mathcal{E}_u is a set of *upper-case edges* [18]:

- Each ordinary edge has the form, $X \xrightarrow{\delta} Y$, representing the constraint $Y - X \leq \delta$.
- Each lower-case edge has the form, $A \xrightarrow{C:u} C$, representing the *possibility* that the contingent duration of link (A, u, v, C) might take on its minimum value, u .
- Each upper-case edge, $C \xrightarrow{C:-v} A$, represents the *possibility* that the contingent duration of link (A, u, v, C) might take on its maximum value, v .

In [11], authors formally defined a *STNU* being *dynamically controllable*, if there exists a strategy for executing the time-points in the network that guarantees that all of the constraints will be satisfied no matter how the contingent durations turn out. Crucially, the durations of contingent links are observed in real-time, as they complete; execution decisions can only depend on past observations.

In [18], authors presented a *polynomial-time* algorithm—called a *MM5 DC-algorithm*—for determining whether any given *STNU* is *dynamically controllable (DC)*. The MM5 DC-algorithm works by recursively generating new edges in the *STNU* graph using the rules shown in Table I. For each rule, pre-existing edges are denoted by solid arrows and newly generated edges are denoted by dashed arrows. Note that each of the first four rules takes two pre-existing edges as input and generates a single edge as its output. In contrast, the Label-Removal rule takes two pre-existing edges as input and replace the upper-case edge with an ordinary one. Finally, applicability conditions of the form, $R \neq S$, should be construed as stipulating that R and S must be distinct time-point variables, not as constraints on the *values* of those variables.

Note that the edge-generation rules only generate new ordinary or upper-case edges. Unlike the upper-case edges in the original graph, the upper-case edges generated by these

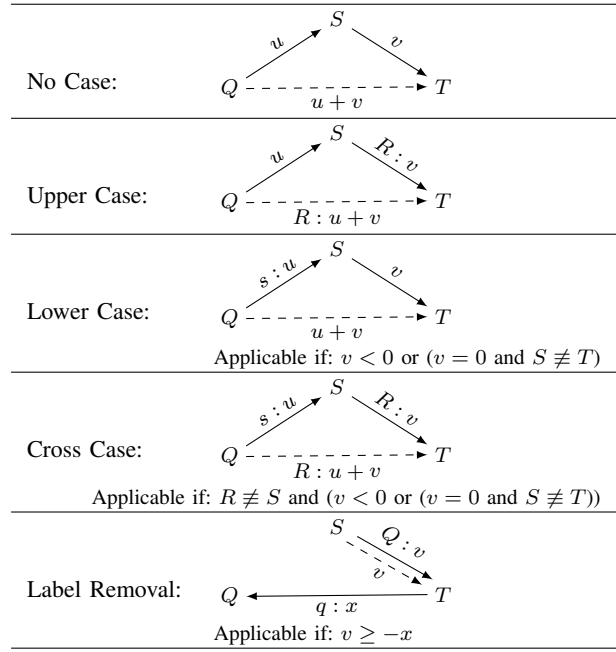


TABLE I
EDGE-GENERATION RULES FOR THE MM5 ALGORITHM. FOR EACH RULE, THE EDGE GENERATED BY THE RULE IS DASHED.

Function MM5-DC-Check(G)

Input: G : *STNU* instance.

Output: the controllability of G .

for 1 to $Cutoff_Bound$ **do**

if (*AllMax* matrix inconsistent) **then return false** ;
 generate new edges using rules from Table I;

if (no edges generated) **then return true** ;

return false

Fig. 9. Pseudocode for the MM5 DC-algorithm. The algorithm returns all minimal bounds of the edges when G is consistent.

rules represent conditional constraints, called *waits* [11].

In particular, an upper-case edge, $Y \xrightarrow{C:-w} A$, represents a constraint that as long as the contingent time-point, C , remains unexecuted, then the time-point, Y , must wait at least w units after the execution of A , the activation time-point for C .

Fig. 9 depicts the pseudocode for the MM5 DC-algorithm. The algorithm performs at most $n^2 + nk + k = O(n^2)$ iterations, which is the number of distinct kinds of edges in a graph having n time-points and k contingent links. In each iteration, the algorithm first computes the *AllMax* matrix—which is the distance matrix for the *Simple Temporal Network (STN)* [31] formed by all of the original and generated, ordinary and upper-case edges (without their alphabetic labels)—and checks that there is no negative cycles in it³ and then applies the rules from Table I to all relevant combinations of edges of the *STNU* from the previous iteration. If no new edges are generated in any given iteration and there is no negative cycle at all, the algorithm

³Morris et al. based their approach on the well known algorithm for *STN* consistency, introduced in [31].

reports that the network is dynamically controllable. If the algorithm continues generating new, stronger edges after the cutoff bound $n^2 + nk + k$, then the network cannot be DC. Since each iteration can be done in $O(n^3)$ time, the overall complexity of the MM5 algorithm is $O(n^5)$.

B. Mapping *TNest* to *STNU*

In this section we show that it is possible to map any *TNest* *wf-path* into an equivalent *STNU* such that the *wf-path* admits a Dynamic Execution Strategy if and only if the corresponding *STNU* is DC. The proof is a constructive one and consists of mapping each *TNest* construct into a corresponding *STNU* fragment.

Theorem 1: Given a *TNest* *wf-path* p , there exists a *STNU* s_p such that p admits a Dynamic Execution Strategy if and only if s_p is DC.

Proof: First of all, we propose the mapping of tasks, edges, basic connectors and *TNest* temporal constraints. Then, we discuss how to deal with the case of a Receive with multiple input streams, that results to be the only connector which requires a particular analysis of one of its possible configurations.

Table II depicts the mapping of *TNest* task, edge, basic connectors and temporal constraints to the associated *STNU* fragments.

– **Task.** Given a *TNest* *wf-path*, each task node T is converted to two *STNU* nodes, T_S and T_E , representing its start and end instants. The duration attribute of T , $[u, v]$, is converted to the contingent link (T_S, u, v, T_E) .

– **Edge.** An edge from task/connector A to task/connector B with delay $[c, d]$ is converted to two constraints between nodes A_E and B_S : $B_S - A_E \leq d$ and $A_E - B_S \leq -c$.

– **Choice connector.** For both start and end Choice connectors the translation is the same. A Choice C with duration attribute $[u, v]$ is converted to two nodes C_S and C_E and two constraints between such nodes: $C_S - C_E \leq v$ and $C_E - C_S \leq -u$.

– **Par connector.** The translation of a start Par connector is similar to that of Choice connectors. The translation of a Par end connector is more subtle. Without loss of generality, in the following, we will consider the case of a Par end connector with two incoming parallel flows. The execution of this connector requires to wait for all incoming flows and, after the last incoming flow occurs, to wait for a time according to the connector range before following the outgoing edge. The key aspect of the connector is that the incoming flows can arrive at different instants, each observing the delay specified in the incoming edge. Therefore, a Par end connector P is represented by two nodes, P_S and P_E and a pair of *buffer* nodes, w_1 and w_2 ; each incoming flow is connected to a *buffer* node and such node to P_S by an edge representing a maximum delay that it could be necessary to wait until the other flow incomes. The constraint between w_1 and P_S waits for the execution of the other buffer node for a maximum time t_1 determined at design time by the controllability-check algorithm (see below). In particular, during the setup of the controllability-check algorithm, the range of the constraint between w_1 and


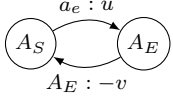
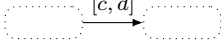
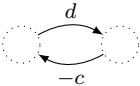
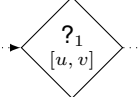
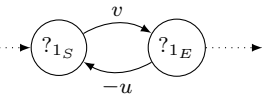
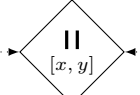
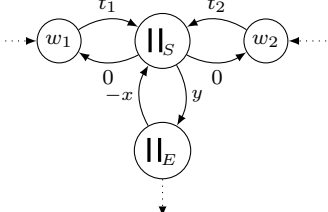
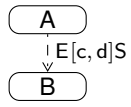
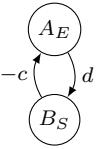
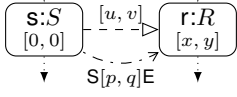
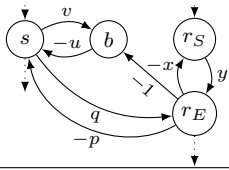
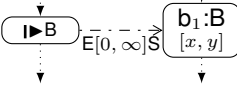
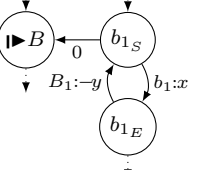
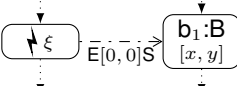
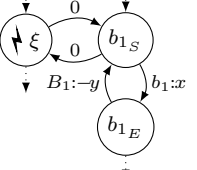
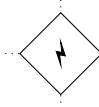

Type	<i>TNest</i> component	<i>STNU</i> fragment
Task:		
Edge:		
Choice:		
Par end:		
Relative constraint:		
Send-Receive:		
Spawn:		
Throw:		
Catch start/end:		

TABLE II
MAPPING OF *TNest* TASK, EDGE, BASIC CONNECTORS AND TEMPORAL CONSTRAINTS TO THE ASSOCIATED *STNU* FRAGMENTS. IN A *STNU*, IF A CONSTRAINT $a - b \leq d$ HAS $|d| = \infty$, THEN THE EDGE REPRESENTING THE CONSTRAINT CAN BE REMOVED.

P_S is set to ∞ and the constraint between P_S and w_1 to 0. Thus, the controllability-check algorithm determines the minimum sufficient value t_1 to allow the synchronization with the other flow in the worst case execution. In the similar way, the upper-bound value t_2 is determined. In this way, if node w_1 is executed before w_2 , the execution *waits* for the occurrence of w_2 before executing P_S as required by the semantics of `Par` end connector. It is simple to show that if there are more incoming flows in the original `Par` end connector, it is possible to translate it using a sequence of pairs of wait nodes properly connected before P_S .

– *Relative constraints.* A relative constraint $\langle I_F \rangle [c, d] \langle I_S \rangle$ between F and S is converted to two constraints between nodes representing the the time points associated to the instants I_F and I_S : $I_S - I_F \leq d$ and $I_F - I_S \leq -c$.

– *Send-receive pattern.* In this pattern, the temporal range $[u, v]$ is the delivery time of a message, while $S[p, q]E$ the validity of a message. In general, at run time a `Send` command can generate messages without any synchronization with the corresponding `Receive` command(s). If one or more messages arrive to a not yet activated `Receive`, they have to be buffered until the command will trigger on and consume them. To properly represent the delivery constraint in the corresponding *STNU*, we have to introduce a *buffer* node to represent the temporal aspect of the possible buffering action. In more detail, `Send` is represented as a single node, since it is a non-blocking activity with a fixed execution time, `Receive` as a pair of nodes, r_S and r_E , linked by two *STNU* constraints representing the duration constraint, and the original delivery constraint as a pair of *STNU* constraints between the send node and a new *buffer* node b representing the buffer stage. The buffer node is then connected to r_E by constraint $b - r_E \leq -1$ to represent the fact the r_E occurs after b . In this way, when a message is sent, b can be activated according to the delivery constraint $[u, v]$. The precedence constraint between b and r_E allows the activation of r_E , after the execution of b , only considering the `Receive` duration constraint $[x, y]$ as expected. If there is a validity constraint in the pattern (i.e., a temporal constraint between `Send` and `Receive`), then it is translated according to its label.

– *Spawn pattern.* In this pattern, the relative constraint $E[0, \infty]S$ guarantees that the `Spawn` command precedes the start of the execution of the associated task. To properly represent the pattern in the corresponding *STNU*, it is sufficient to set a constraint between the node representing the start of the task and the node representing the `Spawn` command with value 0. The `Spawn` command is a non-blocking one and, therefore, one *STNU* node is enough.

– *Throw pattern.* In this pattern, the relative constraint $E[0, 0]S$ guarantees that the `Throw` command immediately precedes the start of the execution of the first task in the exception branch. To properly represent the pattern in the corresponding *STNU*, it is sufficient to set a pair of constraints between the node representing the start of the first task and the node representing the `Throw` command with the same value 0.

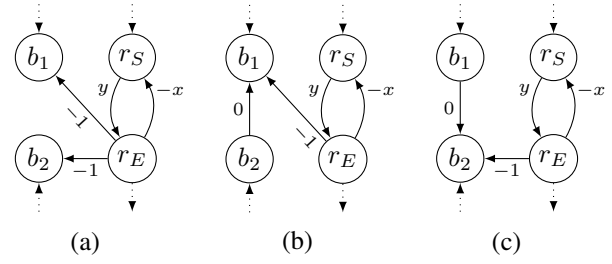


Fig. 10. (a) A *STNU* translation of a `Receive` block with two input streams. (b) a modified *STNU* to determine the relative temporal order between b_1 and b_2 . (c) *STNU* where b_1 occurs before b_2 . (d) *STNU* where b_2 occurs before b_1 .

– *Catch connector.* Any `Catch` connector corresponds to a *STNU* node without any further feature since it is a non-blocking node.

To complete the mapping of all *TNest* blocks, it is necessary now to consider the *multiple Receive pattern*, where a `Receive` command has two or more input streams (cf. Fig. 2(c)). Without loss of generality, in the following, we will consider the case of a `Receive` command with two incoming streams. The behavior of the command with two incoming streams can be described as the behavior of a `Receive` in which the input streams are merged into one: the first incoming message triggers the `Receive`. In other words, it is sufficient that only one `Send` sends a message to trigger (after a suitable time) the `Receive`. The previously described translation of a `Receive` command with one input stream to the corresponding *STNU* can be still considered but with an adjustment. For example, let us consider the *STNU* depicted in Fig. 10(a) where there are two buffer nodes b_1 and b_2 and two nodes r_S and r_E representing the core of the translation of a `Receive` block with two input channels obtained following the method previously described. In order to represent the right behavior, it is necessary to consider two different cases:

- 1) b_1 occurs before (or at the same instant of) b_2 . In this case, *STNU* must have the constraint between b_2 and b_1 representing the precedence of b_1 w.r.t. b_2 and must not have the constraint between r_E and b_2 because b_2 can occur even after r_E . The final *STNU* is depicted in Fig. 10(b).
- 2) b_2 occurs before (or at the same instant of) b_1 . It is the similar configuration of the previous one with b_2 for b_1 and vice versa as depicted in Fig. 10(c).

Therefore, the mapping Fig. 10(a) has to be replaced by Fig. 10(b) and Fig. 10(c) in two different *wf-paths*.

Given a *TNest wf-path* p , applying the above showed mappings to the blocks of p and to the possible relative constraints, it is simple to verify that the obtained *STNU* represents all precedence and temporal constraints but not data flows of the original p .

As formalized in [20], a *STNU* N is said to be *dynamically controllable (DC)* if there exists a *viable dynamic execution strategy*, $S : \Omega \rightarrow T$, from *situations* (*TNest situations* applied to *STNU*) to *schedules* such that for any situations, $\omega', \omega'' \in \Omega$, and executable time-point $x \in N$

it holds that if $[S(\omega')]_x = k$ and $[S(\omega')]_{<k} = [S(\omega'')]_{<k}$, then $[S(\omega'')]_x = k$, where $[S(\omega')]_{<k}$ is the *pre-history* relative to k , i.e., it specifies the durations of all contingent links that execute before x according to the schedule $S(\omega')$. In other words, if the viable strategy S in the situation ω' assigns the value k to the executable time-point x , then S must also assign k to x for any other situation ω'' whose pre-history relative to k matches that of ω' .

Considering the definition of *Dynamic Execution Strategy* for a *TNest wf-path* (cf. Definition 8), it is simple to verify that a *TNest viable dynamic execution strategy* can be easily mapped to a *STNU viable dynamic execution strategy* for the corresponding *STNU* according to the structural mappings presented above. If the *TNest wf-path* has multiple *Receive* patterns, then there exist several corresponding *STNUs*: in this case the *TNest viable dynamic execution strategy* is mapped in a *STNUs viable dynamic execution strategy* for at least one of the corresponding *STNUs* because the set of *STNUs* represent all possible cases that are possible with multiple *Receive* patterns of the original *TNest wf-path*.

Vice versa, given a *STNU* corresponding to a *TNest wf-path* built using the previous mappings, if such *STNU* admits a *viable dynamic execution strategy*, then it is simple to define a *TNest viable dynamic execution strategy* as there is a straight correspondence between temporal constraints of *STNU* and the corresponding ones in *TNest wf-path* induced by construction. If there exist two or more *STNUs* associated to a *TNest wf-path* (this occurs only if *wf-path* contains one or more multiple *Receive* patterns), then it is sufficient that one of *STNUs* admits a *viable dynamic execution strategy* in order to determine the *viable dynamic execution strategy* for the *TNest wf-path*, since it means that at least one relative order among input streams of any multiple *Receive* pattern admits a dynamic execution strategy.

Since the proposed mappings transform each block but multiple *Receive* pattern into a corresponding *STNU* fragment of the same order of size, the execution of the transformation of a *wf-path* can be done in a $O(n)$ time, where n is the number of *TNest* nodes. Then, considering that a *STNU DC* can be checked in time $O(n^5)$ by the MM5 DC-algorithm (cf. Fig. 9), where n is the number of nodes, the overall process to determine a *viable dynamic execution strategy*, if any, of a *TNest wf-path* can be performed in $O(n^5)$ time. ■

C. A Controllability-Check Algorithm

Regarding the determination of a controllability-check algorithm, in [29] a simple exponential-time algorithm, *controllabilityCheck(G)*, has been proposed: it determines whether a given workflow schema G is controllable, its kind of controllability, and duration ranges for connectors/edges (duration ranges of tasks must be leaved unchanged).

Here we propose an optimization of such algorithm that exploits the mapping to *STNUs*. In order to simplify the description of the algorithm, hereinafter we assume that *TNest* schemata have no multiple *Receive* patterns.

Function controllabilityCheck(G)

Input: G : workflow graph to analyze.

Output: the controllability of G and a possibly new set of ranges for activities/delays.

Unroll all possible `Loop` blocks in G ;

WfP = all *wf-paths* of G according also to Fact 1;

/ Initial wf-paths controllability check*

foreach (*wf-path* $p \in WfP$) **do**

R = ranges in p ;

$(status, R) = \text{pathControllabilityCheck}(p, R)$;

if ($status == \text{'Non Controllable Path'}$) **then**

return ($\text{'Non Controllable Wf'}$, \emptyset)

/ All wf-paths are controllable, determine if it is HDC*

?ConnectorSet =

$\{x \mid x \text{ is the last Choice end connector in a wf-path}\}$;

$localStatus = \text{'}$;

foreach ($x \in ?ConnectorSet$) **do**

$(localStatus, R) = \text{controllabilityCheck}(G, x)$;

if ($localStatus == \text{'Prefixes with controllable ranges'}$)

then

$status = \text{'History-Dependently Controllable Wf'}$;

break;

if ($status \neq \text{'History-Dependently Controllable Wf'}$) **then**

$(status, R) = (\text{'Weakly Controllable Wf'}$, \emptyset);

return ($status, R$);

Fig. 11. Algorithm to check the controllability of a workflow graph


Let us call *prefix* of an activity/edge y the set of all *wf-paths* that have the same path from the starting node, , to y . The prefix of y is useful to consider all possible *wf-paths* that can be followed after the execution of y .

Fig. 11 shows the pseudo-code of the proposed algorithm: the algorithm verifies that each single *wf-path* is controllable in isolation and then verifies whether the workflow schema is history-dependently controllable: it considers the (possibly several) last *Choice* end connectors of the workflow schema. *Choice* end connectors are considered as they are responsible of varying the number of *wf-paths* of the schema. Starting from last *Choice* end connectors, the algorithm can verify that for each connector/edge of each prefix there is a suitable range not preventing any possible future execution path in a simpler way than starting from the first *Choice* end connector. This range could be different according to the considered prefix and ranges related to different prefixes could even be disjoint. To do this check, *controllabilityCheck(G)* uses *controllabilityCheck(G, root)*. If the check of each *wf-path* in isolation is negative, the workflow is not controllable. If the check starting from *Choice* end connectors is negative, then the schema is weakly controllable. Otherwise, it is history-dependently controllable.

The algorithm *controllabilityCheck(G, root)*, shown in Fig. 12, firstly determines the prefixes of given activity *root*. For each prefix of *root*, it verifies whether the prefix admits controllable ranges: a prefix has controllable ranges if every connector/edge in the prefix has a unique duration range common to all *wf-paths* of the prefix itself. If all prefixes admit controllable ranges, then it means that *root* may have a temporal range for each of its prefixes but all

Function `controllabilityCheck($G, root$)`
Input: G : a workflow graph; $root$: activity from which to start the analysis
Output: the status of prefix(es) of $root$ and, if prefix(es) admits one, a new set of ranges for activities and delays

P = set of all prefixes of $root$;
 R_p is the array of ranges of activities/edges present in prefix p

```

foreach  $prefix \in P$  do /* Initialize all  $R_{prefix}$  */
  Build  $R_{prefix}$  considering original temporal ranges in  $G$ ;
do
  foreach  $\{prefix \mid R_{prefix} \neq null\}$  do
     $R'_{prefix} = R_{prefix}$ ; /* Save last found solution */
  foreach  $\{prefix \in P\}$  do
    /* Check if the prefix has controllable ranges. */
     $(status, R_{prefix}) =$ 
    buildControllableRanges( $prefix, R_{prefix}$ );
    if  $(status \neq 'Controllable Ranges Found')$  then
      /* It is necessary to evaluate another  $root$  */
      return ('One prefix has not controllable ranges',  $\emptyset$ );
  foreach  $\{a \mid a \text{ is a connector/edge} \wedge a \text{ precedes } root\}$  do
    /* Determine the controllability range for  $a$  among prefixes */
     $P' = \{p' \mid p' \text{ is prefix of } a\}$ ;
    foreach  $prefix' \in P'$  do
       $R_{prefix'}[a] = \bigcap$  all  $a$  ranges in  $wf\text{-paths} \in prefix'$ ;
      if  $(R_{prefix'}[a] == \emptyset)$  then
        return ('One prefix has not controllable ranges',  $\emptyset$ );
      foreach  $\{prefix'' \mid prefix' \leq prefix''\}$  do
        /* Propagate the new range to all prefixes greater than  $prefix'$  */
         $R_{prefix''}[a] = R_{prefix'}[a]$ ;
  while  $(\exists prefix \text{ of activity/edge } b \mid R_{prefix}[b] \neq R'_{prefix}[b])$ ;
  return ('Prefixes with controllable ranges',  $\cup_{prefix} R_{prefix}$ );

```

Fig. 12. Algorithm to check the controllability of a prefix

flows starting from it are possible and controllable. In order to state that the schema is history-dependently controllable, it is necessary to guarantee that each connector/edge preceding $root$ has the same property: in other words, the algorithm has to verify whether for each of connectors/edges preceding $root$ it is possible to derive a common duration range for each of their prefixes.

The derivation of these duration ranges may require to execute several times `buildControllableRanges($prefix, R_p$)` (depicted in Fig. 13) until a fixed-point is reached: indeed, the change of a duration range for a connector/edge may produce other range changes for $wf\text{-paths}$ containing the considered connector/edge.

These changes, in their turn, require that the controllability of modified $wf\text{-paths}$ has to be checked again (through `pathControllabilityCheck(p, R)` depicted in Fig. 14) and it may induce further changes in the duration ranges of connectors/edges.

In the worst case, the time complexity of `controllabilityCheck(G)` can be shown to be

Function `buildControllableRanges(S, R)`
Input: S : set of $wf\text{-paths}$; R : collection of connector/edge temporal ranges
Output: the status and a new set of temporal ranges for connectors/edges

```

do
   $R' = R$ ;
  foreach  $connector \ c$  present into any  $wf\text{-paths}$  do
     $c \text{ range} = \bigcap$  of  $c$  ranges present into  $wf\text{-paths}$ ;
  foreach  $edge \ d$  present into any  $wf\text{-paths}$  do
     $d \text{ range} = \bigcap$  of  $d$  ranges present into  $wf\text{-paths}$ ;
   $R =$  collection of these new ranges;
  foreach  $wf\text{-path } p \in S$  do
     $(status, R) =$  pathControllabilityCheck( $p, R$ );
    if  $(status == 'Non Controllable Path')$  then
      return ('Non Controllable Path Found',  $\emptyset$ )
    if  $(\text{at least one range in } R \text{ is empty})$  then
      return ('Empty Range Found',  $\emptyset$ )
  while  $(\text{at least one range in } R \text{ has changed w.r.t. } R')$ ;
  return ('Controllable Ranges Found',  $R$ )

```

Fig. 13. Algorithm to check the controllability of a set of $wf\text{-paths}$

Function `pathControllabilityCheck(p, R)`
Input: p : $wf\text{-path}$, R : set of connector/edge temporal ranges
Output: the controllability status and, possibly, the new set of temporal ranges for connectors and delays

$s_p = STNU$ associated to p (cf. Theorem 1);
MM5-DC-Check(s_p);
if $(s_p \text{ is not consistent})$ **then**
return ('Non Controllable Path', \emptyset)
else
 /* Determine the new temporal ranges in p from the dynamic ex. st. of s_p */
 $R =$ determine new ranges of p from s_p ;
return ('Controllable Path', R)

Fig. 14. Algorithm to check the controllability of a $wf\text{-path}$

$O(|WfP|^2 n^5 MaxRange^2)$, where n is the number of nodes after the unrolling of all possible `Loop` blocks and $MaxRange$ is the maximum integer value present among the temporal durations/delays. It holds that $|WfP| \leq k^c$, where k is the maximum arity of `Choice` blocks and c is their number.

D. The Complexity of Controllabilities

In the previous section, we introduced the two possible kinds of controllability of a workflow schema. In this section we investigate on the computational complexity of the controllability problem. First of all, we observe that $HDC \Rightarrow WeC$.

1) *Weak Controllability*: There is no possibility other than checking the controllability of each possible $wf\text{-path}$ of a workflow schema separately to state if the schema is weakly controllable. The proof is given as corollary of the following theorem about *coWeak Controllability* (*coWeC*), the complementary problem asking whether a schema contains at least one not controllable $wf\text{-path}$.

Theorem 2: *coWeC* is NP-complete.

Proof: It is sufficient to show the two following properties: i) **coWeC** is in NP and ii) a NP-complete problem is polynomial-time reducible to it.

i) **coWeC** \in NP: as previously discussed, it is possible to check the controllability of a *wf-path* in polynomial time using the function `pathControllabilityCheck()` (cf. Fig. 14). Hence, given a workflow schema and one of its *wf-paths* as certificate it is possible to verify in polynomial time if the *wf-path* is not controllable and, therefore, to state that **coWeC** \in NP.

ii) **SUBSETSUM** \prec_m **coWeC**: given a set of integers I and an integer s , the **SUBSETSUM** problem requires to verify if a non-empty subset $I' \subseteq I$ exists such that the sum of its elements is equal to s . **SUBSETSUM** is NP-complete [32].

To show a polynomial-time many-to-one reduction \prec_m between **SUBSETSUM** and **coWeC**, it is sufficient to set up a polynomial-time function f between the set of instances of **SUBSETSUM** and the set of instances of **coWeC** such that each instance x is a positive instance of **SUBSETSUM** if and only if $f(x)$ is a positive instance of **coWeC** [32]. One possible \prec_m between **SUBSETSUM** and **coWeC** is the following. Given the set $I = i_1, i_2, \dots, i_n$ and the integer s , set up a schema with n sequential **Choice** blocks, each of them associated to an element of I , as depicted in Fig. 15.

Each **Choice** blocks is made by a choice split connector, a task on *true* branch and the choice end connector. Considering the j th **Choice** block, the task on the *true* branch adds the associated element i_j to a global variable S spending i_j time units exactly. Each choice connector choices which branch to follow randomly. After the last **Choice** block, there is another **Choice** block where the equality between the global variable S and the input parameter s is tested. If the S value is equal to $s + 5n$, the task t_s is executed, otherwise nothing occurs. t_s is useful only as milestone to set up the relative constraint with range $[0, s - 1 + 5n]$ from the beginning of the schema. The value $5n$ is the minimum time required to execute all n **Choice** blocks. If the instance of **SUBSETSUM** is positive, then any *wf-path* executing t_s (surely, at least one exists) is not controllable because the time required to calculate S is $s + 5n$ and the relative constraint requires to start t_s within $(s - 1)$ time units after the start of the process. On the opposite direction, given an instance of the schema, if one *wf-path* is not controllable, then it contains the *true* branch of the last **Conditional** split. Hence, the corresponding **SUBSETSUM** instance is positive.

It is simple to verify that the construction of the workflow schema can be done in linear order time with respect to the size of the **SUBSETSUM** instance. ■

Therefore, even the determination of the **WeC** of a schema considering only the *wf-paths* that are really executed requires to check the controllability of all *wf-paths* in the worst case.

Corollary 1: **WeC** problem is coNP-complete.

2) *History-Dependent Controllability:* The *History-Dependent Controllability* definition requires to determine if it possible to perform a *wf-path* satisfying

all relative constraints using allowed delays and allowed connector durations knowing only the durations of connectors/tasks already executed.

HDC problem cannot be in coNP class because, otherwise, it would mean that *coHistory-Dependent Controllability* (**coHDC**) \in NP and this is impossible (unless NP = coNP) since **coHDC** could require to solve **WeC**, already shown to be coNP-complete.

In order to assign **HDC** to a complexity class, let us consider the computational class $\Sigma_2^P = \text{NP}^{\text{NP}}$. It can be described as the class of languages L such that $L = \{x \mid \exists y_1 \forall y_2 \text{ such that } (x, y_1, y_2) \in R\}$, where R is a polynomially balanced, polynomial-time decidable relation and x, y_1, y_2 are strings. A relation $R \subseteq (\Sigma^*)^3$ is polynomially balanced if, whenever $(x, y_1, y_2) \in R$, it holds that $|y_1|, |y_2| \leq |x|^k$ for some k [33].

Examining the definition of fully dynamic execution strategy, it is possible to build a Σ_2^P language L setting the x, y_1, y_2 , and R components as follow:

- x is the string representing a workflow schema TN ;
- y_1 is the string representing a certificate of an execution strategy $st : (WfP_{TN} \times \Omega_{TN}) \rightarrow SC_{TN}$;
- y_2 is the string built concatenating the representation of *wf-paths* p_1, p_2 ;
- R is the relation containing triples (x, y_1, y_2) that satisfies the constraint $\exists st, \forall \omega_1, \forall \omega_2, \forall t \text{ wfHst}([st(p_1, \omega_1)]_t, p_1, \omega_1, st) = \text{wfHst}([st(p_2, \omega_2)]_t, p_2, \omega_2, st) \Rightarrow [st(p_1, \omega_1)]_t = [st(p_2, \omega_2)]_t$. The relation R is a simple extension of the relation verified by the *pathControllabilityCheck* algorithm, In particular, *pathControllabilityCheck* decides (in polynomial time) the relation $R'(p)$ defined as $\exists st', \forall \omega_1, \forall \omega_2, \forall t \text{ sitHst}([st'(p, \omega_1)]_t, p, \omega_1, st') = \text{sitHst}([st'(p, \omega_2)]_t, p, \omega_2, st') \Rightarrow [st'(p, \omega_1)]_t = [st'(p, \omega_2)]_t$. As in section V-A, a *wfHst*(t, p, ω, st) is a *sitHst*(t, p, ω, st) enriched with the set of starting/ending instants for tasks/connectors already executed with respect to time t .

From the above definitions, it is possible to verify that x, y_1, y_2 , and R satisfy the conditions to make L a language in Σ_2^P and that L is the language of **HDC** workflows provided that the execution strategy st' verified by *pathControllabilityCheck* is always the same general defined strategy st required by the definition.

Therefore, **HDC** $\in \Sigma_2^P$. To our knowledge, **HDC** seems not to be Σ_2^P -complete. So far, we know that **HDC** is coNP-hard since **HDC** \Rightarrow **WeC** and **WeC** has been shown to be coNP-complete.

VII. MODELING STEMI GUIDELINES

The *TNest* schema in Fig. 16 summarizes the main process for the diagnosis and treatment of a myocardial infarction (*STEMI*) described in [7]. 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

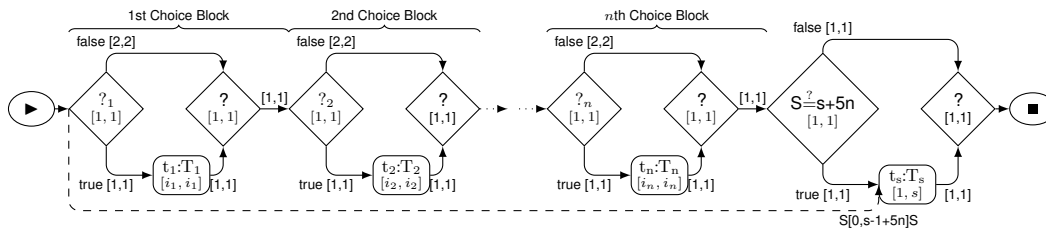


Fig. 15. The workflow schema corresponding to an instance of SUBSETSUM.

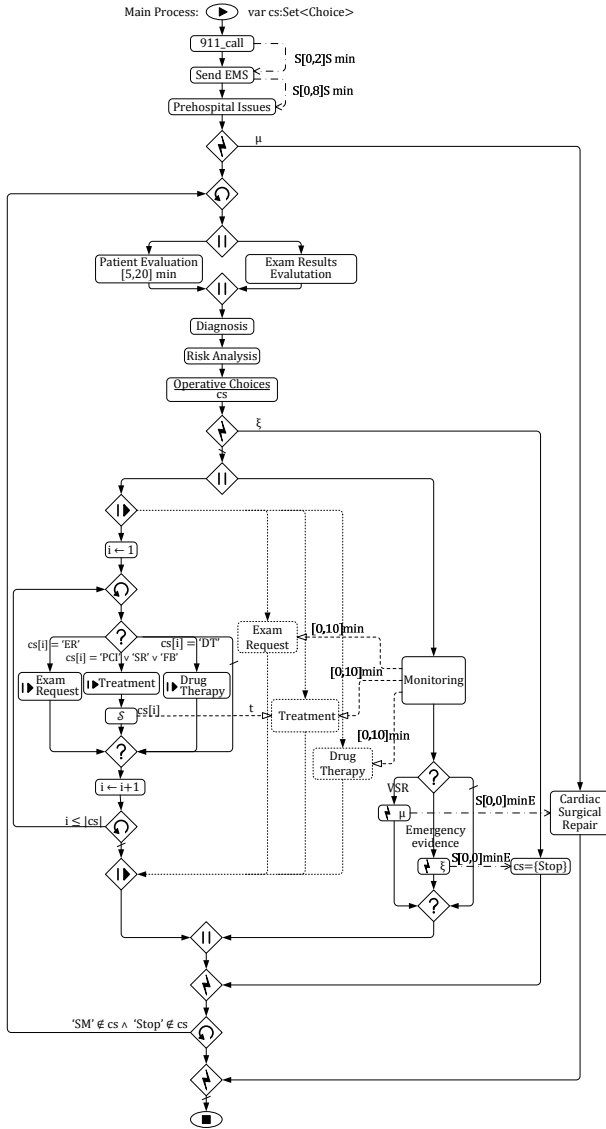


Fig. 16. The main process for the diagnosis and treatment of STEMI. Label ‘SM’ stands for “Secondary Management”, ‘ER’ stands for “Exam Request”, ‘DR’ stands for “Drug Therapy”.

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

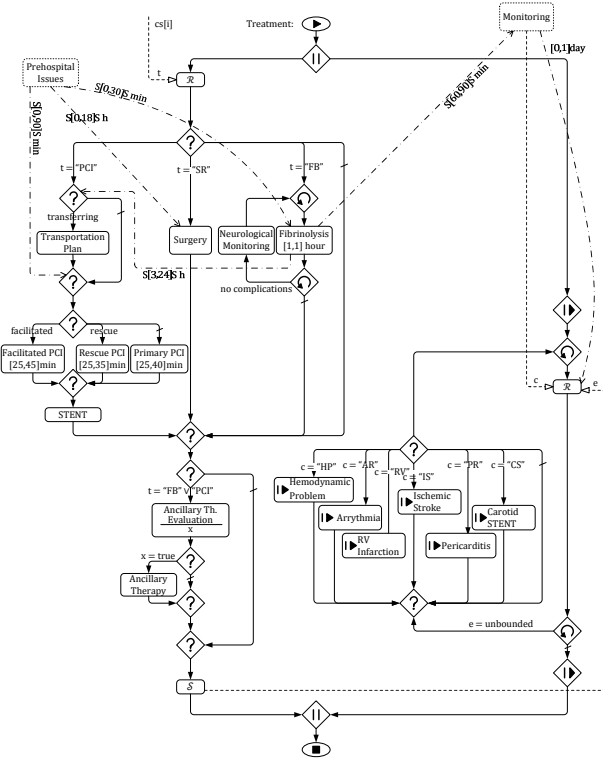


Fig. 17. Details of the process Treatment of Fig. 16.

about the patient vital signs to the operative activities that are executing. Temporal constraints are defined on messages and they specify that patient vital signs have to be communicated within 10 minutes. In case a generic 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. Anyway, if the complication regards an acute rupture of the intraventricular septum (VSR), a different exception is thrown which is handled by the outer `Catch` block by performing an urgent cardiac surgical repair, without any need to further evaluation or diagnosis. The main process terminates when a “secondary management” operative choice is taken.

Process `Treatment` is further specified in Fig. 17: it consists of the main therapeutic actions for the treatment of STEMI. Three main options are possible: the first one, labeled “PCI”, is related to *percutaneous coronary intervention*, the second one, labeled “Surgery”, is related to *coronary artery bypass grafting (CABG) surgery interven-*

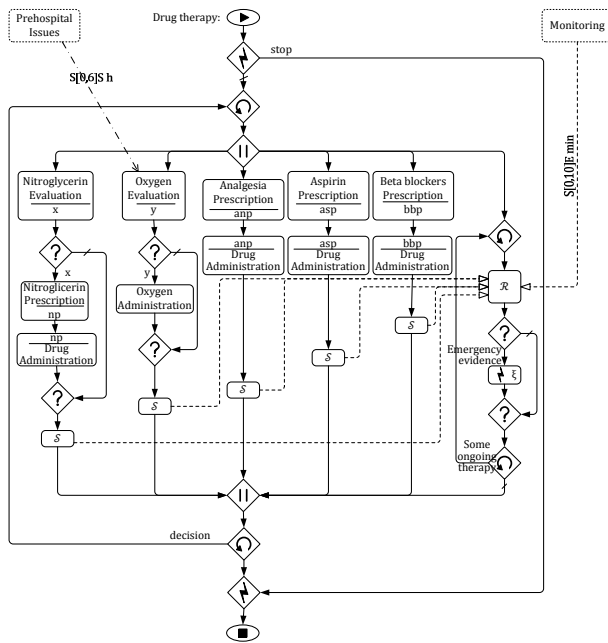


Fig. 18. Details of the process Drug Therapy of Fig. 16.

tion, and the last one, labeled “fibrinolysis”, is related to the administration of fibrinolytic drugs. As you can notice, some temporal constraints have been defined from tasks contained in the main process to tasks in this inner one, and vice versa. In this way, we can represent the main temporal constraints in the *STEMI* guidelines requiring that from the the start of Prehospital issues, Fibrinolysis therapy has to start within 30 minutes, Surgery intervention has to start within 18 hours, and PCI related activities have to start within 90 minutes. For sake of readability, tasks from the main process are represented in Fig. 17 within a dashed-line rounded box. During the therapeutic actions, suitable messages from task Monitoring may activate different tasks for the management of complications, such as *hemodynamic problem*, *arrhythmia*, *RV infarction*, and so on. In Fig. 17, tasks managing such complications are named after the complication name. Many different instances of the Treatment process can be concurrently activated by the loop inside the Concur block in Fig. 16. Therefore, several treatments can be performed in parallel on the patient. Some temporal constraints are specified between treatments activated by different Spawn commands inside subsequent loop iterations.

Process Drug Therapy is further specified in Fig. 18: it consists of the main drug administrations for the treatment of *STEMI*. Different drugs are administered concurrently and several times. The Send blocks S at the end of each drug administration may raise an exception that stops all the other possibly drug administrations in case of emergency without to take any further actions (void exception branch).

VIII. *TNest* AND PROCESS AWARE CLINICAL INFORMATION SYSTEMS

This section briefly describes the role of *TNest* in a modern *WfMS* architecture as well as in the wider setting

of process-aware clinical information systems. A *WfMS* usually provides a multi-user client-server architecture consisting in four main software components: a workflow designer, an engine, a graphical user interface (GUI) and an administration GUI, as depicted in Fig. 19. The *workflow designer* is a client application used to produce a graphical representation of a workflow schema enhanced with auxiliary textual data. Such schema can be translated into an executable specification in a given standard language for a particular run-time system. According to the given environment, the translation can be fully automated or may need several human interventions. In turn, this specification can be interpreted by a software system, usually called *engine*, running on a server. During the workflow execution, the engine can access to additional data needed at run-time (e.g., user roles and user availability for a particular task), and to existing software components exposed as services. The engine is able to run several workflow instances at the same time corresponding to different schemata and versions. The execution of all these instances can be managed through a GUI provided by the engine. Moreover, a specific user interface is associated to each process instance for communicating with end-users. An initial GUI can be automatically generated by the engine starting from the schema, and subsequently customized by developers in order to make it more user-friendly.

TNest is a core formal modeling language that may be used to build a real interpreter which can become the central component of the described *WfMS* architecture. Anyway, *TNest* can also be integrated into an existing system by offering a workflow designer and a translation procedure. The designer should support at least the graphical modeling, validation and simulation of workflow schemata. These features are sufficient to make the language of practical use, for instance during the modeling of medical pathways, since it allows one to formalize complex procedures, share them between different organizations, simulate and check their feasibility, even with respect to temporal constraints.

A further integration step can consist in the realization of a software module for the translation of *TNest* schemata into an existing and standard workflow language, such as BPEL, to take advantage of available technologies. However, a fully automatic translation procedure may be hard to obtain due to the limitations of existing languages and *WfMS*s; for instance, they usually provides a limited support for temporal constraints. Therefore, besides to provide a complete *TNest* interpreter, another viable solution can be the extension of the internal interpreter of an existing *WfMS* engine with the specific constructs of *TNest*.

As for the integration of a *WfMS* based on *TNest* modeling language in a wider architecture considering an health information system, we may point out two different approaches. In the first approach we may consider the *TNest*- based *WfMS* as the core of a process-oriented healthcare information system, where all those clinical processes/pathways requiring a careful management of temporal constraints are represented and managed through the *WfMS*. Integration between data and processes is

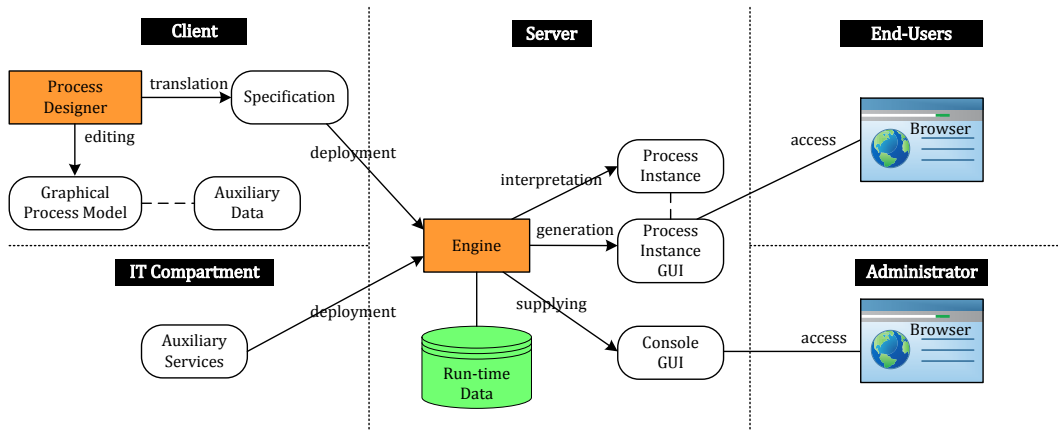


Fig. 19. Architecture of a modern WfMS.

facilitated by the *WfMS* that allows different clinical tasks to synchronize (even through data flows) and suitably share clinical data. In the second approach the overall process-aware health information system may benefit of the special features of *TNest* language: in this case we could think to a *TNest*-based software tool able to check temporal properties of health workflow/process schemata, which are then managed and executed by, for example, the BPEL engine of the health information system. Without loss of generality, we may assume that both these approaches can be realized within a Service Oriented Architecture (SOA), where applications are viewed as software services, described through the WSDL language, able to exchange data between them and to cooperate in a loosely coupled way [34]; indeed, most health information systems provide a large number of SOA services [35], [36] and several SOA-based integrated healthcare systems have been recently proposed [3], [37], [38].

Finally, a further use of a *TNest*-based graphical design tool may a stand-alone one, where physicians can simulate and check different versions and refinements of a clinical pathway, focusing on data-related and critical time features, before entering a clinical pathway in the daily routine. It is worth noting that such use is of particular interest in those clinical domains where time-related requirements are particularly important and complex, as in Intensive Care Units, in follow-up treatments, and so on.

IX. DISCUSSION AND CONCLUSION

In this paper we proposed *TNest*, a new structured, data-centric workflow modeling language for expressing data dependencies, time constraints and exception management in business processes. Data dependencies among tasks are modeled in *TNest* through message passing mechanisms; temporal controllability checking has been proposed for all *TNest* constructs. We formally introduced the concept of *execution strategy* for *TNest* schemata and we proposed two kinds of controllabilities with respect to the possible execution strategies. Then, we proposed a general algorithm that, given a *TNest* schema, determines its kind of controllability.

Moreover, we discussed the computational complexity of the problem of controllability checking.

As a motivating scenario, we considered some important requirements for modeling clinical processes. In particular, we deeply studied the specification of complex guidelines, such as the *STEMI* ones, and the related clinical pathways, for the management of cardiological patients. To this regard, *TNest* allows us to capture some specific requirements from the *STEMI* guidelines that would be difficult to explicitly represent through existing *WfMS*s. Finally, we also discussed how to integrate *TNest* in real world *WfMS* architectures and in clinical information systems.

Future studies will regard the role of *TNest* modularity in the controllability of large workflow schema. In particular, some approximated notion of controllability can be defined in order to take advantage of modular decomposition and reduce the time required by controllability check.

APPENDIX A

TNest FORMAL SEMANTICS

A simplified *TNest* operational semantics is depicted in Fig. 20: it formally describes how each language block is interpreted considering in particular how the process state is updated. As explained in Sec. IV-E, the state of a process is given by the state of its streams, its variables, and the component instances contained in it. In Fig. 20 the process state is denoted by the letter σ eventually enriched with a subscript.

The simplest rule regards the *Sequence* block: let us assume that two tasks A and B have to be executed in sequence, and that the execution of A in the state σ_1 produces a new state σ_2 , while the execution of B in the state σ_2 produces a new state σ_3 , then the execution of the sequence AB starting from the state σ_1 produces the new state σ_3 .

Rules R_{2a} and R_{2b} regards the execution of a *Loop* block with two branches, respectively represented by the composite tasks A and B , and a condition φ defined over a set of variables \bar{x} . If the execution of A in the state σ_1 produces a new state σ_2 in which the condition $\varphi(\bar{x})$ evaluates to false, then the execution of the *Loop* starting from the state σ_1

$$\begin{array}{l}
(R1) \frac{\langle A, \sigma_1 \rangle \rightarrow \sigma_2 \quad \langle B, \sigma_2 \rangle \rightarrow \sigma_3}{\langle \text{seq}(A, B), \sigma_1 \rangle \rightarrow \sigma_3} \quad (R2a) \frac{\langle A, \sigma_1 \rangle \rightarrow \sigma_2 \quad \sigma_2 \vdash \varphi(\bar{x}) \rightarrow \text{false}}{\langle \text{loop}(\varphi, \bar{x}, A, B), \sigma_1 \rangle \rightarrow \sigma_2} \\
(R2b) \frac{\langle A, \sigma_1 \rangle \rightarrow \sigma_2 \quad \sigma_2 \vdash \varphi(\bar{x}) \rightarrow \text{true}}{\langle \text{loop}(\varphi, \bar{x}, A, B), \sigma_1 \rangle \rightarrow \langle \text{seq}(B, \text{loop}(\varphi, \bar{x}, A, B)), \sigma_2 \rangle} \\
(R3a) \frac{\sigma_1 \vdash \varphi(\bar{x}) \rightarrow \text{true} \quad \langle A, \sigma_1 \rangle \rightarrow \sigma_2}{\langle \text{choice}(\varphi, \bar{x}, A, B), \sigma_1 \rangle \rightarrow \sigma_2} \quad (R3b) \frac{\sigma_1 \vdash \varphi(\bar{x}) \rightarrow \text{false} \quad \langle B, \sigma_1 \rangle \rightarrow \sigma_2}{\langle \text{choice}(\varphi, \bar{x}, A, B), \sigma_1 \rangle \rightarrow \sigma_2} \\
(R4) \frac{\langle A, \sigma \rangle \rightarrow \sigma_1 \quad \langle B, \sigma \rangle \rightarrow \sigma_2}{\langle \text{par}(A, B), \sigma \rangle \rightarrow \sigma_1 \cup \sigma_2} \quad (R5) \frac{}{\langle \text{concur}(A, \bar{n}), \sigma \rangle \rightarrow \sigma} \\
(R6a) \frac{\langle A, \sigma_1 \rangle \rightarrow \sigma_2 \quad \sigma_2 \vdash \xi \rightarrow \text{false}}{\langle \text{catch}(\xi, A, B), \sigma_1 \rangle \rightarrow \sigma_2} \quad (R6b) \frac{\langle A, \sigma_1 \rangle \rightarrow \sigma_2 \quad \sigma_2 \vdash \xi \rightarrow \text{true} \quad \langle B, \sigma_2 \rangle \rightarrow \sigma_3}{\langle \text{catch}(\xi, A, B), \sigma_1 \rangle \rightarrow \sigma_3} \\
(R7) \frac{}{\langle \text{skip}, \sigma \rangle \rightarrow \sigma} \quad (R8) \frac{\langle T, \sigma_1 \rangle \rightarrow \sigma_2}{\langle \text{run}(T), \sigma_1 \rangle \rightarrow \sigma_2} \\
(R9) \frac{}{\langle \text{spawn}(T), \sigma \rangle \rightarrow \sigma \cup \sigma_1} \quad (R10) \frac{}{\langle \text{throw}(\xi), \sigma \rangle \rightarrow \xi} \\
(R11) \langle \text{send}(s, \bar{x}, \bar{\alpha}_{out}), \sigma \rangle \rightarrow \sigma[\forall x_i \in \bar{x}. x_i \in \alpha_{out}^i] \\
(R12a) \langle \text{receive}(r, \theta=0, \bar{y}, \bar{\alpha}_{in}), \sigma \rangle \rightarrow \sigma[\exists y_i \in \bar{y}. \alpha_{in}^i y_i \wedge \forall i \neq 1. y_i \text{ unbound}] \\
(R12b) \langle \text{receive}(r, \theta=1, \bar{y}, \bar{\alpha}_{in}), \sigma \rangle \rightarrow \sigma[\forall y_i \in \bar{y}. \alpha_{in}^i y_i \wedge \forall i. y_i \text{ bound}] \\
(R13) \langle \text{empty}(e, \alpha), \sigma \rangle \rightarrow \sigma[\alpha=\emptyset]
\end{array}$$

Fig. 20. A simplified operational semantics of *TNest* constructs.

produces the new state σ_2 . Otherwise, if the condition $\varphi(\bar{x})$ evaluates to true in σ_1 , then the `Loop` execution produces a new state equivalent to the one obtained by sequentially executing task *B* followed by another `Loop` execution. Rule R_{3a} and R_{3b} regarding the `Choice` block can be interpreted in a similar way.

The execution of a `Parallel` block produces a state equal to the union of the states produced by the execution of each single branch: since parallel branches cannot share variables, the produced state are disjoint. Conversely, the execution of a `Concur` block or a `Skip` command do not produce effects on the state, since the first one represents only the scope of a `Spawn` command, while the second one does not perform anything. A `Spawn` command generates a new task instance which is executed in parallel with the other existing ones, hence the effects of such instance execution has to be combined with the state produced by the other ones. While rule R_8 simply states that the effects of a `Run` command are those produced by the task execution.

Rules R_{10} and R_6 regard the exception management: the execution of `Throw` command simply produces a state in which exception ξ is valid. As regards to the `Catch` block, if the execution of the default branch *A* produces a state σ_2 in which the exception ξ is not valid, then the overall block execution produces the state σ_2 ; conversely, if the exception ξ is valid in σ_2 and the execution of the recovery branch *B* starting from σ_2 produces a new state σ_3 , then the overall block execution produces the state σ_3 .

The only two commands that can update the process state are `Send` and `Receive`: in particular, the execution of a `Send` *s* which stores the value contained in a set of variables \bar{x} into the set of output streams $\bar{\alpha}_{out}$, produces a new state equal to the original one but in which the value of each variable $x_i \in \bar{x}$ has been enqueued to the corresponding stream α_{out}^i , as reported in rule R_{11} of Fig. 20. Conversely, the execution of a `Receive` command stores into one variable y_i of \bar{y} the first value contained into one of the available input streams $\bar{\alpha}_{in}$ and sets to unbound the other ones, as reported in rule R_{12a} of Fig. 20. An `and-Receive` has a similar behaviour, but it sets the value of all variables

in \bar{y} . Finally, an `Empty` command produces a state in which the associated stream α is empty.

REFERENCES

- [1] C. Wolf and P. Harmon, *The State of Business Process Management 2012*, ser. A BPTrends Report. Business Process Trends, <http://www.bptrends.com/>, 2012. [Online]. Available: http://www.bptrends.com/members_surveys/deliver.cfm?target=2012-_BPT%20SURVEY-3-12-12-CW-PH.pdf
- [2] R. Lenz and M. Reichert, "It support for healthcare processes - premises, challenges, perspectives," *Data Knowl. Eng.*, vol. 61, no. 1, pp. 39–58, 2007.
- [3] G. Leonardi, S. Panzarasa, S. Quaglioni, M. Stefanelli, and W. M. P. van der Aalst, "Interacting agents through a web-based health serviceflow management system," *J. of Biomed. Informatics*, vol. 40, no. 5, pp. 486–499, 2007.
- [4] R. S. Mans, N. C. Russell, W. M. P. van der Aalst, P. J. M. Bakker, A. J. Moleman, and M. W. M. Jaspers, "Proclerts in healthcare," *J. of Biomed. Informatics*, vol. 43, no. 4, pp. 632–649, 2010.
- [5] P. Gooch and A. Roudsari, "Computerization of workflows, guidelines, and care pathways: a review of implementation challenges for process-oriented health information systems," *J American Medical Informatics Association*, vol. 6, no. 18, pp. 738–748, 2011.
- [6] C. Combi, M. Gambini, and S. Migliorini, "The NestFlow interpretation of workflow control-flow patterns," in *ADBS*, ser. LNCS, J. Eder, M. Bieliková, and A. M. Tjoa, Eds., vol. 6909. Springer, 2011, pp. 316–332.
- [7] E. M. Antman and et al., "ACC/AHA guidelines for the management of patients with ST-elevation myocardial infarction," *Circulation*, vol. 110, no. 5, pp. 588–636, 2004.
- [8] C. Combi and R. Posenato, "Controllability in temporal conceptual workflow schemata," in *BPM*, ser. LNCS, U. Dayal, J. Eder, J. Koehler, and H. A. Reijers, Eds., vol. 5701. Springer, 2009, pp. 64–79.
- [9] C. Combi, E. Keravnou-Papailiou, and Y. Shahar, *Temporal Information Systems in Medicine*, 1st ed. Springer, 2010.
- [10] C. Combi, M. Gambini, S. Migliorini, and R. Posenato, "Modelling temporal, data-centric medical processes," in *Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium*, ser. IHI '12. New York, NY, USA: ACM, 2012, pp. 141–150. [Online]. Available: <http://doi.acm.org/10.1145/2110363.2110382>
- [11] P. H. Morris, N. Muscettola, and T. Vidal, "Dynamic control of plans with temporal uncertainty," in *IJCAI*, B. Nebel, Ed. Morgan Kaufmann, 2001, pp. 494–502.
- [12] R. Laue and J. Mendling, "The Impact of Structuredness on Error Probability of Process Models," in *UNISCON*, 2008, pp. 585–590.
- [13] H. Reijers and J. Mendling, "Modularity in Process Models: Review and Effects," in *BPM*, 2008, pp. 20–35.
- [14] B. Kiepuszewski, A. ter Hofstede, and C. Bussler, "On Structured Workflow Modelling," in *CAiSE*, 2000, pp. 431–445.
- [15] C. Combi, M. Gozzi, R. Posenato, and G. Pozzi, "Conceptual modeling of flexible temporal workflows," *ACM Trans. Auton. Adapt. Syst.*, vol. 7, no. 2, pp. 19:1–19:29, Jul. 2012. [Online]. Available: <http://doi.acm.org/10.1145/2240166.2240169>
- [16] A. Lanz, B. Weber, and M. Reichert, "Time patterns for process-aware information systems," *Requirements Engineering*, pp. 1–29, 2012. [Online]. Available: <http://dx.doi.org/10.1007/s00766-012-0162-3>
- [17] T. Vidal and H. Fargier, "Handling contingency in temporal constraint networks: from consistency to controllabilities," *J. Exp. Theor. AI*, vol. 11, no. 1, pp. 23–45, 1999.
- [18] P. H. Morris and N. Muscettola, "Temporal dynamic controllability revisited," in *AAAI*, M. M. Veloso and S. Kambhampati, Eds. AAAI Press, 2005, pp. 1193–1198.
- [19] P. Morris, "A structural characterization of temporal dynamic controllability," in *CP*, ser. LNCS, F. Benhamou, Ed., vol. 4204. Springer, 2006, pp. 375–389.
- [20] L. Hunsberger, "Fixing the semantics for dynamic controllability and providing a more practical characterization of dynamic execution strategies," in *TIME*, C. Lutz and J.-F. Raskin, Eds. IEEE, 2009, pp. 155–162.
- [21] —, "A fast incremental algorithm for managing the execution of dynamically controllable temporal networks," in *TIME*, N. Markey and J. Wijsen, Eds. IEEE, 2010, pp. 121–128.

- [22] D. Wang, M. Peleg, S. W. Tu, A. A. Boxwala, R. A. Greenes, V. L. Patel, and E. H. Shortliffe, "Representation primitives, process models and patient data in computer-interpretable clinical practice guidelines: a literature review of guideline representation models," *I. J. Medical Informatics*, vol. 68, no. 1-3, pp. 59–70, 2002.
- [23] S. Quaglini, M. Stefanelli, A. Cavallini, G. Micieli, C. Fassino, and C. Mossa, "Guideline-based careflow systems," *Art. Intell. Med.*, vol. 20, no. 1, pp. 5 – 22, 2000. [Online]. Available: <http://www.sciencedirect.com/science/article/B6T4K-410D1C7-2/2/ebb21b31f60b303e8b62fe8f77fc26a0>
- [24] C. Combi, M. Gozzi, J. M. Juárez, B. Oliboni, and G. Pozzi, "Conceptual modeling of temporal clinical workflows," in *TIME*. IEEE, 2007, pp. 70–81.
- [25] C. Combi, M. Gozzi, B. Oliboni, J. M. Juárez, and R. Marín, "Temporal similarity measures for querying clinical workflows," *Artificial Intelligence in Medicine*, vol. 46, no. 1, pp. 37–54, 2009.
- [26] M. P. Fanti, A. M. Mangini, M. Dotoli, and W. Ukovich, "A three-level strategy for the design and performance evaluation of hospital departments." *IEEE T. Systems, Man, and Cybernetics: Systems*, vol. 43, no. 4, pp. 742–756, 2013. [Online]. Available: <http://dblp.uni-trier.de/db/journals/tsmc/tsmc43.html#FantiMDU13>
- [27] W. M. P. van der Aalst, L. Aldred, M. Dumas, and A. H. M. ter Hofstede, "Design and implementation of the YAWL system," in *CAiSE*, ser. LNCS, A. Persson and J. Stirna, Eds., vol. 3084. Springer, 2004, pp. 142–159.
- [28] C. Combi and M. Gambini, "Flaws in the flow: The weakness of unstructured business process modeling languages dealing with data," in *OTM Conf.*, ser. LNCS, R. Meersman, T. S. Dillon, and P. Herrero, Eds., vol. 5870. Springer, 2009, pp. 42–59.
- [29] C. Combi and R. Posenato, "Towards temporal controllabilities for workflow schemata," in *TIME*, N. Markey and J. Wijsen, Eds. IEEE, 2010, pp. 129–136.
- [30] —, "On the complexity of temporal controllabilities for workflow schemata," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ser. SAC '12. New York, NY, USA: ACM, 2012, pp. 60–66. [Online]. Available: <http://doi.acm.org/10.1145/2245276.2245292>
- [31] R. Dechter, I. Meiri, and J. Pearl, "Temporal constraint networks," *Artif. Intell.*, vol. 49, no. 1-3, pp. 61–95, 1991.
- [32] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.
- [33] C. M. Papadimitriou, *Computational complexity*. Addison, 1994.
- [34] M. P. Papazoglou and W.-J. van den Heuvel, "Service oriented architectures: approaches, technologies and research issues," *VLDB J.*, vol. 16, no. 3, pp. 389–415, 2007.
- [35] A. Wright, D. W. Bates, B. Middleton, T. Hongsermeier, V. Kashyap, S. M. Thomas, and D. F. Sittig, "Creating and sharing clinical decision support content with web 2.0: Issues and examples," *Journal of Biomedical Informatics*, vol. 42, no. 2, pp. 334–346, 2009.
- [36] K. Kawamoto, A. Honey, and K. Rubin, "Model formulation: The hl7-omg healthcare services specification project: Motivation, methodology, and deliverables for enabling a semantically interoperable service-oriented architecture for healthcare," *JAMIA*, vol. 16, no. 6, pp. 874–881, 2009.
- [37] D. M. Lopez and B. Blobel, "A development framework for semantically interoperable health information systems," *I. J. Medical Informatics*, vol. 78, no. 2, pp. 83–103, 2009.
- [38] S. Jeong, C.-H. Youn, E. B. Shim, M. Kim, Y. M. Cho, and L.-M. Peng, "An integrated healthcare system for personalized chronic disease care in home-hospital environments," *IEEE Transactions on Information Technology in Biomedicine*, vol. 16, no. 4, pp. 572–585, 2012.
- [39] N. Markey and J. Wijsen, Eds., *TIME 2010 - 17th Int. Symp. on Temporal Reprs. and Reas.*, Paris. IEEE, 2010.



Carlo Combi received the Laurea Degree in E.E. from the Politecnico of Milan in 1987. In 1993, he received the Ph.D. degree in biomedical engineering. Currently, he is Full Professor of Computer Science at the Dept. of Computer Science, University of Verona, Italy. From July 2009 to June 2013 he was chair of the Artificial Intelligence in Medicine Society (AIME). Main research interests are related to the database and information system field, with an emphasis on the management of clinical data and processes.



Sara Migliorini received the Laurea degree in Computer Science from the University of Verona in 2007. In 2012, she received the Ph.D. degree in Computer Science from the same university. Currently, she is Postdoctoral Research Associate at the Dept. of Public Health and Community Medicine, University of Verona, Italy. Main research interests are related to the database and information system field, with an emphasis on the management of geographical data and processes.



Mauro Gambini received the Laurea degree in Computer Science from the University of Verona in 2007. In 2012, he received the Ph.D. degree in Computer Science from the same university. Currently, he is Postdoctoral Research Associate at the Dept. of Public Health and Community Medicine, University of Verona, Italy. Main research interests are related to the database and information system field, with an emphasis on the management of business processes.



Roberto Posenato received the Laurea degree in Computer Science and the Ph.D. in Computational Mathematics from the University of Milan, in 1991 and 1996, respectively. Currently, he is Assistant Professor of Computer Science at the Dept. of Computer Science, University of Verona, Italy. Main research interests are related to the graph algorithms and combinatorial optimization problems, with an emphasis on the management of business processes.