A formal approach to the analysis of clinical workflow languages

M. Adela Grando Ph.D.¹, David Glasspool Ph.D¹ and John Fox Ph.D²

¹School of Informatics, University of Edinburgh, 10 Crichton Street, Edinburgh, EH8 9AB, UK
²Department of Engineering Science, University of Oxford, Parks Road, Oxford, OX1 3PJ, UK

Address correspondence to:

M. Adela Grando
10 Crichton Street, School of Informatics, University of Edinburgh,
Edinburgh, UK, EH8 9AB
Tel: +44 (0) 131 651 5657
Fax: +44 131 650 6899
Abstract

Objective: To develop proof strategies to formally study the expressiveness of workflow-based languages, and to investigate their applicability to clinical workflow languages.

Method: We propose two strategies for studying the expressiveness of workflow-based languages based on a standard set of workflow patterns expressed as Petri Nets and notions of congruence and bisimilarity from process calculus. Proof that a Petri Net-based pattern $P$ can be expressed in a language $L$ can be carried out semi-automatically. Proof that a language $L$ cannot provide the behavior specified by a Petri Net $P$ requires proof by exhaustion based on analysis of cases and cannot be performed automatically. The proof strategies are generic but we exemplify their use with a particular clinical workflow language, PROforma. To illustrate the method we evaluate the expressiveness of PROforma against three standard workflow patterns and compare our results with a previous similar but informal comparison.

Results: We show that the two proof strategies are effective in evaluating a workflow language against standard workflow patterns. We find that using the proposed formal techniques we obtain different results to a comparable previously published but less formal study. We discuss the utility of these analyses as the basis for principled extensions to workflow languages. Additionally we explain how the same proof strategies can be reused to prove the satisfaction of patterns expressed in declarative languages based on linear temporal logic.

Conclusion: The proof strategies we propose are useful tools for analysing the expressiveness of clinical workflow languages. This study provides good evidence of the benefits of applying formal methods of proof over semi-formal ones.

Keywords: Petri Net, process calculus, design patterns, clinical workflows
Introduction

A number of languages have been developed to support the computerization of clinical workflows [4]. Recently there has been a growing interest within the field of health informatics in applying formal methods to the study, analysis and comparison of clinical workflows, see for instance the Protocure project [5]. At the level of deployment of clinical workflows formal methods have been used for comparing the care given to the patient with the recommendations of medical guidelines, checking whether physicians’ actions follow a recommended path in the clinical workflow [5]. For verifying the satisfaction of properties in clinical workflows theorem proving and model checking have been used [5]. For studying the expressiveness of workflow-based languages Petri Nets (PNs) [6] and Process Algebra have proved to be very popular formalisms, see for instance [1,3,9, 10,11,17,18]. Petri Nets is a formalism for modeling Discrete Event Dynamic Systems (DEDS) where the concurrency and competition of shared resources play a special role. While having a very formal semantic Petri Nets are also provided with a graphical representation based on directed bipartite graphs. Process algebras are a family of related approaches to formal specification of concurrent systems. For instance the pi-calculus [2] is a type of process algebra. Using process algebra it is possible to provide a high-level description of interactions, communications, and synchronizations between a collection of independent processes. They also provide algebraic laws that allow manipulation and analysis of process descriptions and permit formal reasoning about congruences and similarities between processes.

In this paper we investigate the use of PNs as a formal framework for studying the expressiveness of workflow-based languages. In Section II.B we introduce an extension of PNs called Colored Petri Nets (CPNs) [7]. In Section II.C we discuss a useful set of 43 “prototypical” control flow patterns introduced by van der Aalst et al [1] and expressed as CPNs. The patterns have also been translated to process calculus [7] [8]. Originally these patterns were introduced for sharing and reusing a range of control flow dependencies typically encountered in workflow modeling. Later the patterns have been used for checking the expressiveness of workflow languages and performing comparative studies of expressiveness between languages. They have
A formal approach to the analysis of CIG languages, Grando et al also been translated to process calculus [7] [8]. The patterns can also potentially act as a set of baseline requirements against which workflow or process based technologies could be designed or extended. While the patterns provide a potential basis for formal and rigorous analysis they have mainly been used in informal comparative studies of workflow languages [1][3][9] [10] [11].

While the patterns provide a potential basis for formal and rigorous analysis they have mainly been used in informal comparative studies of workflow-based languages [1][3][9] [10] [11]. In [3] Mulyar et al showed that the patterns from [1] also recurrently occur in clinical workflows which implies that the patterns are relevant for the health informatics community, and that it should be advantageous for clinical workflow languages to support those patterns. Mulyar et al [3] have informally compared the expressiveness of the clinical workflow languages Asbru [12], GLIF [13], EON [14] and PROforma [15] with respect to the patterns [1]. In this paper we build on Mulyar et al’s work [3] by showing how formal analysis techniques may be applied in the comparison of language expressiveness.

In Section III we introduce two strategies that can be used to prove that a workflow language can or cannot express a particular workflow pattern expressed as a CPN.

Proof that a pattern can be expressed is approached by showing that a workflow fragment expressed in the language can be formally mapped to a CPN which is congruent to the target pattern, according to the bisimulation properties defined in pi-calculus by Milner [16]. In more general terms this strategy can be used to prove that two workflows defined in different languages are congruent. To the best of our knowledge no formal strategy has been applied to prove the satisfaction of patterns for workflow languages in general, and languages for the specification of clinical workflows in particular.

Proof that a pattern can be expressed is approached by showing that a workflow fragment expressed in the language can be formally mapped to a Petri Net which satisfies one of the simulation equivalences defined in the pi-calculus by Milner [16] with respect to the target pattern. In more general terms this strategy can be used to prove that two workflows defined in different languages satisfy some bisimilarity relationship. Proof that a language cannot express the behaviour described by a workflow pattern is approached through analysis by cases. This
A formal approach to the analysis of CIG languages, Grando et al

strategy is non-automatic and it only differs from similar informal methodologies used for
instance in Mulyar et al work [3] in the requirement of an exhaustive search to prove that there is
no possible combination of components from the language that can provide the behaviour
described by the pattern. This methodology is not error-free because no mechanism to check the
exhaustivity of the analysis has been provided.

The strategies that we present in Section III are generic enough to be applied over any workflow-
based language. In order to exemplify the use of these strategies we chose the PROforma
language, with which we have particular experience. As we explain in Section II.A PROforma is
representative of a typical clinical workflow language, but more importantly it is one of the
languages studied in Mulyar et al’s work [3] and we wish to compare their results directly with
ours. Other potential candidates for exemplifying these strategies could have been Asbru, GLIF
and EON, also studied in [3]. As an introduction to computer-based medical guidelines and
protocols we recommend the survey of Ten Teije et al. [].

In Section IV we apply the strategies developed in Section III to the PROforma language. Using
the first strategy we prove that PROforma fully satisfies pattern 24 (Persistent Trigger) from van
der Aalst et al’s patterns [1]. In [17] we have already proved that PROforma does satisfy pattern
10 (Arbitrary cycles). Using the second strategy we prove that PROforma does not satisfy pattern
5 (Simple Merge) and pattern 8 (Multi-Merge). For space reasons we could not include the
proofs of: non-satisfaction of pattern 14 (multiple instances with a priori run-time knowledge),
non-satisfaction of pattern 15 (multiple instances without a priori run-time knowledge) and the
partial satisfaction of pattern 23 (Transient trigger). These formal results partially differ from the
informal analysis presented in Mulyar et al’s study [3] where it was stated that PROforma
satisfies pattern 5, does not satisfy pattern 10 and fully satisfies pattern 23. The results that we
obtain demonstrate the need to use formal analysis techniques to study language features. A
particular benefit in the case of the study of language expressiveness is that close examination of
the reasons for the inability of a language to express a particular pattern provides a principled
basis for proposing extensions to the language. We discuss the lessons learned from the failure of
PROforma to express patterns 5 and 8.

While CPN based patterns have been very popular, other formalisms have been proposed for
expressing workflow patterns. In Section V we explain how to reuse the proof strategies
A formal approach to the analysis of CIG languages, Grando et al presented here for proving that a workflow-based language satisfies patterns expressed in DECLARE [], a declarative formalism whose formal semantic is given in linear temporal logic.

Finally in Section VI we present our conclusions and we propose future directions of research.

I. **Background**

A. **The PROforma Language**

While the CPN control flow patterns proposed in van der Aalst et al [1] can be used to study the expressiveness of any workflow-based language, here we exemplify their use with the PROforma language.

PROforma is an executable process modelling language that has been successfully used to build and deploy a range of decision support systems, clinical workflows and other clinical applications. It has a declarative syntax and a well-defined operational semantics, described by Fox et al [15]. PROforma is one of several clinical workflow modelling languages which are broadly comparable in scope although they take somewhat different approaches to process representation, as compared by Peleg et al [18].

An obvious question is why such languages are needed at all, given the availability of well-developed formalisms such as CPNs and Process Algebra which can support formal analysis as well as process description. There are two main reasons. Firstly, clinical workflow modelling languages like PROforma are designed to be more accessible to users who do not have a background in computer science. The abstractions they provide are intended to be more comprehensible to clinicians. Secondly, clinical workflow modelling languages typically offer additional features over and above simple process and data modelling. For example, PROforma incorporates a well-developed decision model based on argumentation logic, intended to provide the abstractions and the mechanism to enable complex clinical decisions to be concisely and clearly modelled.

A PROforma clinical workflow consists of a small set of task classes that can be composed into networks representing arbitrarily complex plans or procedures, and a similarly small set of
attributes which control task enactment. A *keystone* represents a very abstract notion of task with a non-specific implementation. The keystone task type is specialized into four more concrete task types – *action*, *enquiry*, *decision* and *plan* - which are used to compose concrete clinical workflows (optionally it is also possible to leave a task in a workflow specified only in abstract terms as a keystone, and defer the choice of a specific task to enact it until runtime). The four concrete class types are used as follows. Actions represent procedures to be executed on the external environment (e.g. administering a drug or updating a database). Enquiries are tasks carried out to acquire information from the user (for instance choose the dose for a drug), from the application (data definitions and variables associated with the PROforma workflow which are used to specify the tasks’ attributes: for instance the constraint that a task can only be enacted if variable Q is greater than 0) or from some external system (hospital patient medical records, ontologies of medical knowledge). Decisions are processes for making choices about what to believe or what to do. Plans are collections of tasks grouped together for some reason, perhaps because they share a common goal or use a common resource, or need to be done at the same time. All the types of tasks share a small set of common attributes which they inherit from the root *keystone* class, including:

- **State:** which indicates the state of execution of the task during workflow enactment. A task is *dormant* if the PROforma engine has not yet considered executing it, a task is *discarded* if the engine has explicitly decided not to execute it, a task is *in_progress* if it is currently being executed, and a task is *completed* if its execution has finished. Normally a task starts in state *dormant*, and ends up *completed* or *discarded*. Optionally a task may be defined as *cyclic*, in which case it is allowed to transition from *completed* back to *in_progress* under defined conditions so that it may be executed multiple times.

- **Precondition:** a condition that must be satisfied for the task to start execution. If the *precondition* is not satisfied but all of the other constraints described below are satisfied, then the task is *discarded*.

- **Antecedent_tasks:** a set of task identifiers that indicates the tasks that must be completed before this one starts. Essentially this is a special purpose *precondition*. 
• **Trigger:** an externally introduced message allowing tasks to be explicitly started without waiting for their scheduling constraints to be satisfied. During the enactment of a plan an external message can be introduced multiple times and each time it is introduced the associated tasks are (re)started. If there is more than one task with the same trigger, all of them are simultaneously activated.

• **Wait_condition:** a condition that, when true, allows a task to be started without waiting for its scheduling constraints to be satisfied (if its preconditions are satisfied it will be enacted, otherwise it will be discarded). If there is more than one task with the same wait_condition, all of them are simultaneously activated. While a trigger can activate a task multiple times, a wait_condition can only activate a task once during the execution of a (non-cyclic) plan.

• **Postcondition:** an assignment statement that is assumed to be executed when the task completes.

• **Nbr_cycles_expression:** an expression that when it is evaluated it indicates the number of time the task should be cyclically enacted.

• **Cycle_until:** indicates that a task should be cyclically enacted until this expression evaluates true.

• **Parent plan:** if the task is part of a plan, then this identifies that plan.

• **Confirmatory:** whether or not the task needs to be confirmed by user intervention before it is allowed to execute.

Besides the above attributes shared by all task types, each particular type of task has some additional specific attributes. For instance tasks of type plan have a termination_condition and an abort_condition such that when the first condition is satisfied the plan immediately becomes completed and when the second condition is satisfied the plan is immediately discarded.
The PROforma execution engine changes the state of a keystone by evaluating the following functions:

*InitialiseConditions*: if this function evaluates to true the state of the task becomes *dormant*. This function returns true when the task state is *completed* if the task has been triggered, or if the task is cyclic and a new cycle should be started, or the *InitialiseConditions* of the task’s parent plan are true.

*StartConditions*: if this function evaluates to true the state of the task can be changed from *dormant* to *in_progress*, and in the case of cyclic tasks the state can also change from *completed* to *in_progress*. This function returns true iff, the task’s parent plan (if any) is *in_progress* and either its *ScheduledStartConditions* are true and it has no *trigger*; or it has been triggered or is *completed* and needs to start a new cycle.

*DiscardConditions*: if this function evaluates to true the state of the task can be changed to *discarded*. It returns true iff any of the following are true:

- **A.1.** The task is either *dormant*, *in_progress* or waiting out a cycle repeat interval and the *DiscardConditions* or *TerminationConditions* of its parent plan are true, or

- **A.2.** It is currently *dormant*, its parent plan is *in_progress*, its *ScheduleConditions* are true, and either it has antecedent tasks that have all been *discarded*, or it has a *precondition* that is not true, or

- **A.3.** The task is a plan and its *abort conditions* are true.

*CompleteConditions*: if this function returns true then the state of the task can be changed from *in_progress* to *completed*. The *CompleteConditions* of a task are true if it is currently *in_progress* and if its completion has been confirmed by the user (if it is a *confirmatory* task that requires user confirmation), or all the data sources have been supplied (if it is a decision or enquiry), or if at least one of its terminal components (components that are not in the list of *antecedent_tasks* of any task) has *completed* (if it is a plan) and if none of its components is *in_progress* or could subsequently become *in_progress* or be *discarded* (if it is a plan).
A formal approach to the analysis of CIG languages, Grando et al

**ScheduledStartConditions**: this function evaluated to true iff the task is *dormant*, its
**ScheduleConditions** are true, at least one of its *antecedent_tasks* (if it has any) has *completed* and its *precondition* (if any) is true.

**ScheduleConditions**: is true iff all the task’s *antecedent_tasks* have either been *completed* or been *discarded*, and if the *wait_condition* (if any) is true.

**TerminationConditions**: of a plan are true if either its *termination_condition* exists and evaluates true or if it has a terminal task which has *completed*.

In figure 1 we present an example of a PROforma plan. The plan starts with the general practitioner (GP) performing the action of meeting the patient and it continues with the enactment of the keystone corresponding to measuring the patient’s blood pressure. This task is represented as a keystone because at run time the GP can decide to take the blood pressure himself or to delegate the task to a practitioner nurse. The keystone represents the goal of measuring the patient’s blood pressure rather than the procedure by which it will be achieved. After the patient’s blood pressure has been measured it can be recorded in the patient’s medical record, which is accessed by an enquiry, and the GP can decide the patient’s diagnosis. Based on the diagnosis two different plans can be enacted to provide appropriate treatment.

*Figure 1 about here*

In PROforma it is not possible to concurrently enact the same plan specification for different patients, but multiple instances of an activity can arise during a plan execution in four ways:

**Cycles**: we can define cyclic tasks that can be sequentially executed \(n > 0\) times, where \(n\) can be defined during design time or run time. For defining cyclic tasks two attributes can be specified for the number of cycles (*nbr_cycle_expression*) and for conditioning the loop (*cycle_until*). These attributes can be specified in terms of variables whose value is fixed at design or run time.

**Triggers**: these allow an activity to be initialized during the enactment of the parent plan each time it receives an external *trigger*. 
Copying (cloning): we can define \( n > 1 \) copies of the same activity in the specification of a plan, where \( n \) is known at design time. Those copies have different identifiers and can be executed independently and concurrently during plan enactment.

Recursive definition: a plan \( P \) is recursively defined when its definition contains a copy of itself.

B. Colored Petri Nets

Colored Petri Nets (CPNs) [6] are a high-level Petri Net variant which extends the PN formalism to allow different types of tokens to be distinguished. Different types of token are graphically represented as dots of different colors, and the term “color” is used to indicate the type of a token – for example we speak of tokens of “color” PATIENT in our example below. Besides having an exact mathematical definition of their execution semantics, CPNs are provided with a graphical notation based on directed bipartite graphs where the nodes represent transitions (i.e. discrete events that may occur) and places (i.e. conditions), and the nodes are connected by directed arcs (that describe which places are pre- and/or postconditions for which transitions). The evolution of the system is determined by the triggering of an enabled transition, which removes a fixed number of colored tokens from its input places and adds a fixed number of colored tokens into its output places (according to the cardinality of its input/output arcs).

In figure 2 we give an example of a CPN for simulating the workflow of a radiologist. The radiologist selects a patient from the patients that are waiting to have their X-ray taken, he takes the patient’s X-ray and once he develops the X-ray he hands it to the patient. The patient then leaves and the radiologist is free to take another patient. The CPN is specified using the Colored Petri Net tool [19]. Places are graphically represented with ovals, transitions are drawn as boxes and arcs are represented with arrows. Arcs connect places with transitions and transitions with places. When arcs connect places with transitions they can have an associated condition. For instance in figure 2 patient waiting is a place, Meet patient is a transition and the arrow that connects them is an arc without condition.

The CPN formalism specifies for each place a domain of permitted colors, in terms of the Cartesian product of all colors in the model. In figure 2 the place’s permitted color is written
A formal approach to the analysis of CIG languages, Grando et al. below it in capital letters. For example the place patient waiting contains tokens of color PATIENT, the place radiologist free contains tokens of color RADIOLOGIST and the place pre Xr contains tokens of color PATIENTxRADIOLOGIST, where the color PATIENTxRADIOLOGIST represent pairs of tokens where the first element in the pair is of color PATIENT and the second element in the pair is of color RADIOLOGIST.

For a transition to be able to occur there must be sufficient correctly colored tokens available to match its condition. The transition’s labeled incoming arcs indicate the number and color of the tokens required, and the transition’s labeled outgoing arcs indicate the number and color of the tokens produced. For instance the transition Meet patient can occur if there is a token of color PATIENT in the place patient waiting (as indicated by the variable $p$ of type PATIENT that labels the incoming arc) and one token of color RADIOLOGIST in the place radiologist free (as indicated by the variable $r$ of type RADIOLOGIST that labels the outgoing arc). The CPN shown in figure 2 is initialized with three tokens of type PATIENT in place patient waiting, and one token of type RADIOLOGIST in place radiologist free. A binding is made between the tokens chosen and the variables used in the condition. When a transition is enabled the set of tokens that were bound on incoming arcs are removed from the corresponding input-places and a set of tokens is added to each output-place. For example when the transition Meet patient is fired it removes a token with color PATIENT from the place patient waiting and it removes a token with color RADIOLOGIST from the place radiologist free. With the removed tokens a new token is created with color PATIENTxRADIOLOGIST, which is added to the place pre Xr (as indicated by the pair $(p,r)$ of type PATIENTxRADIOLOGIST).

CPNs can simulate nondeterministic and concurrent behaviour. From a set of $m > 0$ enabled transitions $n$ of them, $n \leq m$, can be chosen to be simultaneously executed. Additionally for a given transition multiple bindings can occur concurrently iff there are enough colored tokens available for the chosen bindings. For instance in figure 2 the radiologist can choose to meet (enact transition meet patient) any of the three patients that are waiting for the x-ray (any token from place patient waiting).

[Figure 2 about here]
C. Colored Petri Nets for specifying control flow patterns

In the area of workflow management a large number of competing languages are available and the relations between them in terms of expressiveness are usually not very clear. Even in the same language there is sometimes variability, because the language provides different mechanisms to specify the same functionality with different workflows. CPNs have been proposed by van der Aalst et al in [1] as a formal framework for studying the expressiveness and variability of workflow-based languages. The basis of the work presented in [1] is a useful set of 43 patterns represented as CPNs that describe control-flow (or process) fragments in terms of activities and their execution ordering through different constructors, which permit flow of execution control, such as sequence, choice, parallelism, join synchronization, etc. The patterns range from fairly simple constructs present in almost all workflow languages to complex routing primitives that few languages can provide.

The patterns are divided into a number of categories: basic control-flow patterns, advanced branching and synchronization patterns, structural patterns, multiple instance patterns, state-based patterns, cancellation patterns and “new control flow patterns”. In Table 1 we provide an informal description of each of the categories.

[Table 1 about here]

The patterns have proved to be very popular in the community of developers and users of workflow and process based technologies [1,3,9, 10,11,17,18] for comparing and evaluating the expressiveness and adequacy of existing languages and tools. For instance showing that a language is unable to express a pattern demonstrates the language’s weaknesses and suggests a future language extension in order to provide the missing functionality. For this reason the patterns have inspired or directly influenced the principles on which new workflow or process based technologies have been designed.

While the patterns provide a potential basis for formal and rigorous analysis, they have mainly been used in informal comparative studies. To date the strategy used to justify why a language satisfies one of the CPN patterns proposed by van der Aalst et al[1] consists of providing a process specification in the language and a natural language explanation of its conformance to
A formal approach to the analysis of CIG languages, Grando et al

the pattern, while proof of non satisfaction consists of providing an informal justification explaining the reasons why the language does not conform to the pattern. In [1] the expressiveness of fourteen workflow systems (Staffware, WebSphere, MQ, COSA, iPLanet, SAP Workflow and FileNet), case handling systems (FLOWer), business process execution languages (BPMN, UML 2.0 Activity-Diagrams and EPCs) and business process execution languages (BPEL4WS, WEbSphere BPEL, Oracle BPEL and XPDL) have been analyzed. In Mulyar et al [3] the expressiveness of the clinical workflow languages Asbru [12], GLIF [13], EON [14] and PROforma [15] has been studied with respect to these patterns. For each language considered in Mulyar’s work [3] they checked if it was possible to realize the control flow patterns from [1] with the facilities offered by the language. In the Conclusions we analyze how our results differ from theirs.

I. Methodology

In this section we propose two formal strategies to study the expressiveness of workflow-based languages. The strategies are based on the control flow patterns of [1] and on notions of bisimilarity from process algebra.

C. Strategy for proving that a language provides the behavior specified by a CPN

A language $L$ fully (partially) satisfies a pattern $P$ if there exists a workflow $W$ specified in language $L$ such that $P$ and $W$ are congruent for all (some) variants of $P$.

To prove that a pattern $P$ can be expressed in a language $L$ the following steps are followed:

1. Specify a workflow $W$ in language $L$, which could potentially provide the behavior described by $P$. The difficulty of completing this step depends on: (a) the user’s understanding of the pattern $P$, (b) the user’s experience modeling clinical workflows in language $L$ and (c) the simulation facilities provided by the tool that allow the user to enact the specified workflow to check if it behaves as the pattern $P$. The methodology presented here does not advise the user on how to find $W$, but allows formal proof that $W$ behaves like the pattern $P$. 
2. Translate the workflow $W$ into an equivalent CPN form. The idea of specifying clinical workflows as Petri Nets is not new, take for instance our previous work [17], Quaglini et al’s work [20] or that of Peleg et al. [21]. If language $L$ has been provided with an algorithm to map it into CPN this step is automatic.

3. Use the Colored Petri Net tool [19] over the pattern $P$ and the CPN resulting from 2) to automatically compute their state space graphs. A state space graph is a finite state transition system that describes all the possible flows of execution generated by the CPN.

4. Use the Edinburgh Concurrency Workbench tool [22] over the state space graphs obtained in 3) to automatically determine if they satisfy any of the bisimilarities specified in process calculus by Milner [16]. If it is not possible to prove any congruence then it means that workflow $W$ does not implement pattern $P$.

Here we will consider four variants of bisimilarity, two of them (trace equivalence and strong bisimilarity) based on assuming that all tasks in a workflow are observable, and the other two (weak bisimilarity, branching bisimilarity) based on differentiating between tasks with observable behavior and internal unobservable tasks. For full formal definitions of the bisimilarity properties checked by the Edinburgh Concurrency Workbench tool the reader is referred to the documentation for the tool, available at []. Rather than repeat the lengthy formal definitions here it will be more useful to informally describe each of the bisimilarity properties:

a) Trace equivalence: the simplest of the bisimilar relationships, it requires that two processes generate the same set of execution traces (that is, the same set of possible sequences of execution over their component tasks) independently of their internal specification. For instance if we take the CPNs shown in figure 3(a) and 3(b), they are trace equivalent because they generate the same set of execution traces: task $A$ followed by task $B$ or followed by task $C$ (i.e. traces $A.B$ and $A.C$).

b) Strong bisimilarity: this requires that all tasks in a pair of process have externally observable behaviour, and that this behaviour is indistinguishable. The CPNs shown in figure 3(a) and 3(b) are not strong bisimilar, because in the case figure 3(b) after the execution of task $A$ there is still the possibility of choosing between task $B$ and $C$, while in the CPN in figure 3(a) this option is no longer available once task $A$ has been enacted
A formal approach to the analysis of CIG languages, Grando et al

(if the first copy of A is enacted then only B is possible, while if the other copy of A is enacted then only C is possible).

c) Weak bisimilarity (observation equivalence): this differentiates between tasks with external observable behaviour and internal tasks (denoted as λ) without observable behaviour. For instance a task that requires data from a user is a task with external observable behavior, while a task that collects data from databases is an internal task. In order to “mirror” an observable task in a process it is possible to precede and follow the task with a number of internal tasks in the bisimilar process. This equivalence does not preserve branching structures, which means that decision points in the workflows will not necessarily coincide. For example if we take the CPNs shown in figure 3(c) and 3(d) and we assume that tasks A and B have observable behavior and task λ is an internal task, they are weak bisimilar.

d) Branching bisimilarity: this holds when a weak bisimilarity additionally preserves the branching structure of the process by further imposing that all the internal tasks executed before an observable task is enacted lead to states that offer identical sets of possible future choices, and similarly all the internal tasks executed after an observable task must lead to states that offer identical sets of possible future choices (though not necessarily the same set of future choices as before the observable task was enacted).

If we take the CPNs shown in figure 3(d) and 3(e) and we assume that tasks A and B have observable behavior and task λ is an internal task, they are not branching bisimilar because in 3(e) the internal task λ executed before task B leads to a state that does not offer the choice of selecting task A, while in figure 3(d) this option is available before enacting task B.

Strong bisimilarity and branching bisimilarity define congruences, weak bisimilarity and trace equivalence do not. If two processes are congruent then they can be freely interchanged. But if two processes are not congruent then they can be interchanged only if they are not part of, or composed with, other processes. For example if we take a workflow W1 with subflow P1 and a workflow W2 with subflow P2, then if P1 is not congruent with P2 there is no guarantee that after interchanging P1 with P2 the workflows W1 and W2 will behave as before. If P1 and P2 are congruent, then after interchanging them W1 and W2 will have identical behaviour.
The formal methodology introduced here and the informal methods that have been used to prove the satisfaction of patterns, for instance in Mulyar et al [], share in common step (1) the proposal of a candidate workflow $W$ in language $L$ for the behavior described by pattern $P$. What our methodology adds in order to construct the formal proof of pattern satisfaction are steps (2) to (4) which can be performed automatically if language $L$ is provided with an algorithm to map any workflow in $L$ into an equivalent CPN. Therefore for languages which have been provided with mappings to some PN variant, like PROforma [ProHealth], GLIF[Mor] and GUIDE[Quaglini-based careflow systems], this methodology should be easily applicable. In section IV we explain how we implemented this methodology in the PROforma language without encountering any major difficulty.

While the notions of bisimilarity explained above are very useful for the study of the expressiveness of languages, in general the focus during workflow modeling is on the workflow’s goals rather than control flow. Consequently even if two workflows defined in the same language do not have a bisimilar control flow, they may still be interchangeable if, based on some formal verification technique (like model checking), they prove to achieve the same goal or intention.

**D. Strategy for proving that a language does not provide the behavior specified by a CPN**

In our previous work [17] we have been concerned with proving that a workflow pattern specified in CPN form can be expressed in a particular workflow language. Here we also consider the analysis of the patterns that cannot be expressed in the language. Such an analysis is useful in identifying specific features of the language which block the expression of these patterns, and may therefore suggest principled extensions to the language.

In order to prove that a language $L$ cannot provide the behavior specified by a CPN $P$ we must prove that there is no possible combination of components from $L$ that can provide the behaviour described by pattern $P$. Our strategy in this case is to carry out a proof by exhaustion based on an analysis of cases. Such a proof has two stages. We must first show that some critical aspect of $P$
A formal approach to the analysis of CIG languages, Grando et al.
can only be modeled in $L$ in a finite set of different ways, characterized as distinct and well-defined cases. For each of these cases we must then show that the approach in $L$ cannot be applied to $P$. The main disadvantages of this strategy are firstly that it cannot be performed automatically and secondly that the results are not guaranteed to be error-free. The methodology does not provide any mechanism to check that the considered cases are all the possible cases of interest and that no significant case has been overlooked. There is the possibility that in the future a different case can be found that invalidates the proof.

II. Exemplification of the methodology: formal study of the expressiveness of PROforma based on control flow patterns

In this Section we examine a number of control flow patterns from van der Aalst et al [1] which can or cannot be expressed in PROforma. Whereas in the CPN patterns the notion of transition corresponds to an abstract notion of task, for the PROforma plans introduced throughout the proofs in this Section we use the equivalent notion of keystone to represent an abstract task.

1) Basic control-flow patterns

Of the first set of control-flow patterns in [1] (the so-called “basic” patterns), PROforma does not satisfy patterns 5 and 8. These patterns are closely related, and we consider first the non-satisfaction of pattern 8 because the proof for pattern 5 is easily derived from it.

Pattern 8 (Multi-Merge): this pattern specifies convergence of two or more branches of workflow into a single subsequent branch. Each time an incoming branch is enabled the thread of control is passed to the subsequent branch. Figure 4 (a) illustrates the pattern with two incoming branches. If the pattern is initialized with one token in place $I_1$ and one token in place $I_2$ then all the possible traces of execution are: $A.C.B.C$, $B.C.A.C$, $A.B.C.C$ and $B.A.C.C$. To give a more concrete example: immediately after either analysis A or analysis B is completed (tasks $A$ or $B$), the patient needs to collect the results (task $C$). Even if $A$ and $B$ are simultaneously performed, the results for each analysis need to be independently collected.

[Figure 4 about here]
We prove here that PROforma does not satisfy pattern 8 for the case of two incoming branches, but the proof can be generalized to $n \geq 0$ incoming branches. The critical issue for PROforma is that if both incoming branches are executed, task C must be executed twice. As noted above there is a well-defined and limited set of cases in which a PROforma task within a single clinical workflow may be executed more than once. We have to consider all the cases that could allow task C to be executed multiple times. In PROforma tasks and plans can either be cyclic or non-cyclic. Within a non-cyclic plan two executions of task C could be achieved either by having a single instance of C execute twice, or by having two instances of C each executed once. If on the other hand we consider a cyclic plan, we can achieve two executions of C by allowing the plan to cycle twice. We can therefore divide the problem into three cases:

**Case 1)** The plan is non-cyclic and there is one instance of the task C that can be activated twice. This case is represented by plan P1 from figure 4 (b), where all the components correspond to generic tasks or PROforma keystones. In this situation there are only four means by which task C may be activated.

Firstly tasks A and B may be defined as *antecedent_tasks* of C. According to function *StartConditions* C will become *in_progress* once all its antecedents have completed (as specified by function *ScheduleConditions*). This clearly will not allow the desired pattern to be implemented regardless of the issue of activating C multiple times.

Secondly task C can be provided with the *precondition* “Completed(A) or Completed(B)”. This avoids the need to specify antecedent tasks for C. However, because it has no antecedent tasks defined, task C will now be discarded before it can be executed, because after the *parent_plan* becomes *in_progress* the function *DiscardConditions* returns true, because function *ScheduledConditions* evaluates true but the *precondition* of C is false.

Thirdly in the absence of antecedent tasks being defined, according to function *StartConditions* (as specified by *ScheduledStartConditions*) task C may be activated by a *wait_condition* when it is not cyclic, has no *trigger* and has no *antecedent_tasks* or *precondition*. However *wait_conditions* in PROforma are limited to only allow activation of a task only once during the execution of a (non-cyclic) plan. So even if we add the attribute *wait_condition* = “Completed(A) or Completed(B)” to task C, this task can never be executed more than once.
A formal approach to the analysis of CIG languages, Grando et al

Finally according to function StartConditions a trigger could activate C. Triggers in PROforma were introduced to allow arbitrary triggering of sections of workflow by external events, and are not subject to the limitation of one activation per execution of the plan. A trigger could therefore activate C multiple times. However triggers may only be introduced externally, they may not be programmatically generated by tasks within the workflow. Triggers may not therefore be used to implement this pattern.

Case 2) The plan is non-cyclic and there are two copies of task C. As we show with plan P2 of figure 4 (c) even if we are able to generate all the traces specified by the pattern, according to function Complete Conditions we still have to wait for the completion of both instances of task C in order to complete plan P2 and therefore set in_progress any task that contains plan P2 as antecedent_task.

In plan P2 we introduce two copies of keystone C with identifiers C1 and C2 respectively. We specify as antecedent_tasks of C1 the task A, and as antecedent_tasks of C2 task B. In this case we can generate all the traces specified by pattern 8. However pattern 8 requires that each execution of task C follows a unique thread of control. The intent of the pattern is that there should be a single exit point leading to subsequent workflow. Equivalently in PROforma this corresponds to the possible enactment of a task that contains plan P2 as antecedent_task every time a copy of task C is completed. But according to function CompleteConditions plan P2 can be completed only when both copies of task C are completed.

Case 3) The plan is non-cyclic and task C is cyclic with two iterations. As shown in plan P1 in figure 4 (b) once task C is enacted and it becomes completed, according to function StartConditions, it is not possible to avoid its consecutive enactment, allowing the traces of execution A.C.C.B or B.C.C.A.

Case 4) The plan is cyclic with two iterations. Consider plan P1 of figure 4(b) defined as a cyclic plan with two iterations. The workflow contained in P1 will be executed twice, a single branch of the plan being allowed to execute on each occasion. All possible execution traces are A.C.B.C, B.C.A.C, A.C.A.C or B.C.B.C. However the latter two traces are not permitted by
A formal approach to the analysis of CIG languages, Grando et al

pattern 8, while two traces that are permitted \((A.B.C.C \text{ and } B.A.C.C)\) cannot be generated. Moreover according to function \textit{CompleteConditions} a cyclic plan introduces the restriction that the flow of execution can only continue after two executions of \(C\) are completed, while pattern 8 allows the flow to continue after each execution of \(C\). A cyclic plan \(P1\) cannot therefore be considered.

We conclude that PROforma does not provide any mechanism that can activate \(C\) both after the execution of \(A\) and after the execution of \(B\), and therefore is unable to implement pattern 8.

**Pattern 5 (Simple Merge):** this pattern is closely related to pattern 8. It describes the convergence of two or more branches into a single subsequent branch. Each enablement of an incoming branch results in the thread of control being passed to the subsequent branch. This pattern provides a way of merging two or more branches without synchronizing them, therefore there is no need to explicitly replicate a sequence of activities that is common to the branches. Instead the common set of activities need only to be depicted once, thus in the case presented in figure 4(a) only traces \(A.C.B.C\) and \(B.C.A.C\) are possible.

For example: analyses \(A\) and \(B\) both require the presence of the patient, so they can be performed independently but not simultaneously. After each analysis the results need to be collected (task \(C\)).

The restriction that tasks \(A\) and \(B\) cannot be simultaneously executed can be straightforwardly implemented in PROforma by the use of a single mutually exclusive variable which is accessed by PROforma enquiry tasks. The variable is set by tasks which are antecedents of \(A\) and \(B\) and released by tasks executed after both \(A\) and \(B\). However the additional restriction does not affect any of the cases considered for pattern 8, so the above analysis still applies and pattern 5 is not implementable in PROforma for the same reasons.

**What do we learn from the analysis of pattern 5 and pattern 8?**

The proofs that PROforma does not satisfy patterns 5 and 8 raise two separate issues. Firstly in the current PROforma definition the semantics associated with the attribute \textit{antecedent_tasks} are quite restrictive. If we define for a task \(X\) the tasks \(Y_1, \ldots, Y_n\) as \textit{antecedent_tasks}, then we
restrict the execution of $X$ to only be possible once all of tasks $Y_1, \ldots, Y_n$ are either completed or discarded. An obvious extension to the semantics of the antecedent tasks attribute would be to allow its definition in terms of conjunctions and disjunctions of predicates “Completed($X$)” and “Discarded($X$)” where $X$ is a task. For example we would like to arrange an appointment with the patient when the prescribed drug treatment has been completed or if treatment has been suspended due to medication intolerance.

Secondly in PROforma there is no efficient way to merge multiple executions into a single flow of execution. The only way to model these situations is to provide a copy of the subsequent plan for each branch opened by a decision point. For example if pattern 8 has $n$ incoming tasks $A_1 \ldots A_n$ and a synchronizing task $C$, then $n$ copies of the task $C$ with precondition= Completed($A_1$) or..... or Completed($A_n$) should be specified to achieve the same expressiveness.

2) “New” control flow patterns

In this Section we consider the second set of control flow patterns of van der Aalst et al [1], the so-called “new” patterns comprising pattern 21 to pattern 43. Of these, we prove here that PROforma satisfies pattern 24. Pattern 24 is concerned with the activation of tasks by asynchronously generated signals ("triggers").

**Pattern 24 (Persistent Trigger):** in this pattern execution of a specific activity instance within a specific process instance is triggered by an internally or externally supplied asynchronous signal. If the activity cannot be executed at the time the trigger is introduced, then the trigger is not lost but it is placed in a queue to allow the later activation of the activity.

There are two variants of this pattern. The first variant is presented in the first CPN of figure 5(a) where transitions $Produce\_trigger$ and $A$ can be fired multiple times, generating one token each time they are fired. In this pattern multiple triggers can be buffered in a place $trigger$ until activity $B$ is enabled. Once $B$ is enabled (task $A$ has been fired and there is at least one token in the buffer $trigger$) it executes consuming the token generated by task $A$ and a single token from $trigger$. Examples of valid execution traces are: $Produce\_trigger.A$. $B$. and $A.Produce\_trigger.B$. While $A.B.Produce\_trigger.B$ cannot be generated by the pattern because task $B$ cannot be enacted after task $A$ has been executed if no trigger has been fired.
The second variant is shown in the second CPN of figure 5(b) corresponding to triggers as initiators of flows of execution. In this case the initialization of the activity $B$ does not depend on the execution of other previous activities, but only depends on the trigger fired by the transition $Produce\ trigger$. An example of valid execution is: $Produce\_trigger.B$. While $B.Produce\_trigger$ cannot be generated by the pattern because task $B$ cannot be enacted before a trigger has been fired.

Example: a nurse can take a blood sample from a patient when the physician requires it during consultation, or on regular basis for a hospitalized patient under monitoring. If the nurse is available a blood sample can be taken when requested, but if the nurse is busy the requests are put in a queue and once the nurse is free they can be responded to.

[Figure 5 about here]

In pattern 24 a task will be enacted if its scheduled start condition is true and it has been triggered by a persistent trigger. As we explained in Section II.A according to function $StartConditions$ in PROforma a non cyclic task can be initiated by a trigger or wait\_condition, independently of its scheduling constraints. Below we prove that PROforma satisfies pattern 24. We prove that we can implement in PROforma persistent triggers and that we can restrict the execution of a task to the satisfaction of its scheduling constraints and the presence of a trigger or the satisfaction of a wait\_condition.

To implement persistent triggers in PROforma we proceed by defining an auxiliary task with an associated trigger or wait\_condition, such that each time the task is activated by the event or wait\_condition it increments an integer variable $Q$ representing the number of triggers that have not yet activated any task. We define $Q$ as a variable with mutually exclusive access so that it can only be accessed by one task at a time. Tasks can access variable $Q$ by activating tasks of type enquiry.

Additionally we define task $B$ with wait\_condition “$Q>0$” and postcondition “$Q:=Q-1$”. In this way $B$ will be executed only if its scheduling conditions are satisfied and the auxiliary task has been triggered.
A formal approach to the analysis of CIG languages, Grando et al

To prove that this approach allows the implementation of pattern 24 we need to analyze four cases, corresponding to the two variants of the pattern (figure 5(a) and figure 5(b)) and internal and external origins for the trigger.

Firstly we consider the first pattern variant, where the trigger is an internal signal. We define the PROforma plan \( P1 \) from figure 5(c) where the internal trigger is simulated by a \( \text{wait\_condition} \). The task \( \text{produceTrigger}(PT) \) accesses the enquiry \( E \) to get exclusive access to variable \( Q \) and has \( \text{postcondition} \ Q := Q + 1 \) to record triggers fired. Task \( B \) accesses the enquiry \( F \) to get exclusive access to variable \( Q \) and has an associated \( \text{wait\_condition} = \text{“Q>0”} \). Therefore according to function \( \text{StartConditions} \) (in particular \( \text{ScheduleConditions} \)) execution of task \( B \) remains pending until task \( A \) has been performed and \( Q > 0 \). Task \( B \) has \( \text{postcondition} \ Q := Q - 1 \) to record the consumption of a trigger. Subplans \( P \) and \( S \) are infinite cycles, and according to function \( \text{CompleteConditions} \) plan \( S \) cycles after the execution of \( B \).

Secondly we consider the first pattern variant, but now the trigger corresponds to an external signal. For this case we can specify the PROforma plan \( P1 \) from figure 5(c) as described above where the task \( \text{produceTrigger}(PT) \) has associated a trigger \text{start\_thread} to simulate the external trigger. According to function \( \text{StartConditions} \) task \( B \) becomes \text{in\_progress} when \text{start\_thread} is fired.

Thirdly we consider the second pattern variant, where the trigger is an internal signal. We choose the PROforma plan \( P2 \) from figure 5(d). It differs from plan \( P1 \) only in the removal of task \( A \) because in this variant the trigger initiates the execution of task \( B \). Then according to function \( \text{StartConditions} \) the execution of task \( B \) remains pending until \( Q > 0 \), i.e. until an internal trigger has been introduced.

Finally it remains to consider the second pattern variant, where the trigger is an external signal. For this case we define the PROforma plan \( P2 \) from figure 5(d) as explained in 3 where the task \( \text{produceTrigger}(PT) \) has associated the trigger \text{start\_thread}.

For each of these four cases we must prove that the corresponding PROforma plan is congruent with pattern 24. We only provide here the proof for case 3 corresponding to the plan \( P2 \) of figure 5(d). The proofs for the other cases are very similar and may easily be derived from this proof. As explained in Section III.A) the proof consists on:
A formal approach to the analysis of CIG languages, Grando et al

1. Specify a workflow $W$ in language $L$ to provide the behavior described by pattern $P$: the plan $P_2$ in figure 5(d) (workflow $W$) specified in PROforma (language $L$) was introduced above as a candidate for the second variant of pattern 24 where the trigger is an internal signal (pattern $P$).

2. Translate the workflow $W$ into an equivalent CPN form. This step corresponds to mapping the PROforma plan $P_2$ of figure 5(d) into a CPN. We present in figure 6(d) the CPN resulting from applying the algorithm that we presented in [17] to the PROforma plan $P_2$. For simplicity we limited in the CPN the number of triggers to 3, instead of allowing unlimited triggers (the proof generalizes to the case of unlimited triggers). Because the algorithm of [17] does not provide any mapping for data, after applying the algorithm we added the places $Queue$, $LoopSubplan1$ and $LoopSubplan2$ to save the tokens that encapsulate respectively the variables $Q$ and $cycle$. The variable $cycle$ was introduced to record the number of iterations of subplans $P$ and $S$.

Use the Colored Petri Net tool [19] over the pattern $P$ and the CPN resulting from (2) to automatically compute their state space graphs. For us pattern $P$ is the second variant of pattern 24 (figure 6(b)) and the CPN obtained in 2 (figure 6(d)).

3. Use the Edinburgh Concurrency Workbench tool [22] over the state space graphs obtained in (3) to automatically determine if the PROforma plan $P_2$ in figure 5(d) satisfies some bisimilarity with the second variant of pattern 24. The tool returned the following results:

a.a) Trace equivalence: ignoring actions without visible effect both nets generate the same execution traces. To demonstrate this we assumed that except for tasks $B$ and $ProduceTrigger (PT)$ the rest of the tasks had no visible effects. Because we have restricted our proof to three tokens the execution traces generated by both space graphs were given by the following regular expression: $PT.B.PT.B.PT.B + PT.PT.B.PT.B + PT.PT.PT.B.B.B + PT.B.PT.PT.B.B + PT.PT.B.PT.B$. For instance the expression $PT.PT.PT.B.B.B$ corresponds to the case in which task $B$ is performed three times immediately, after the introduction of three triggers.

a.b) Strong bisimilarity: this property was not satisfied.

a.c) Weak bisimilarity: this property was not satisfied.
A formal approach to the analysis of CIG languages, Grando et al

a.d) Branching bisimilarity: this property was satisfied: both nets have the same branching structure.

Branching bisimilarity defines a congruence, therefore the PROforma language satisfies the second variant of pattern 24.

Our experience with this proof was that the implementation of every step required by the methodology was relatively easy and fast, except for step (2) which was more time-consuming. We estimate that this proof (steps 2-4) could be reproduced in 3 hours.

[Figure 6 about here]

What do we learn from the analysis of pattern 24?

Pattern 24 in van der Aalst et al [1] treats internal and external triggers uniformly such that they satisfy the following properties:

Property 1: A trigger can enable a task multiple times while the workflow is active, creating multiple instances of the task.

Property 2: A trigger can only enable a task instance if the scheduling constraints of the task instance are satisfied.

Property 3: A trigger can enable only one instance from a set of active tasks instances.

In PROforma the notions of trigger and wait_condition explained in Section II. A. resemble the notion of trigger from [1] whose properties are enumerated above, but with some differences. In PROforma Property 1 is satisfied only for triggers, because only triggers are able to activate the same task more than once while the plan is active. Wait_conditions can only initiate a task once while the plan is active.

Property 2 is directly satisfied only by wait_condition, but for triggers Property 2 can be simulated. In PROforma a trigger activates a task instance regardless of the satisfaction of its scheduling constraints. But as we showed with pattern 24, in PROforma it is possible to simulate Property 2 for triggers by introducing an auxiliary task.

Property 3 is not satisfied for either triggers or wait_conditions. In PROforma when a trigger or wait_condition is introduced all the task instances that can be activated are simultaneously
A formal approach to the analysis of CIG languages, Grando et al.

activated. The reason for this is that PROforma has not been designed with enactment of different concurrent and independent instances of the same plan specification in mind. This is an important limitation of PROforma. If in future PROforma extensions the concurrent enactment of clinical workflows were to be incorporated, this would force us to select a semantic for trigger and wait_condition which allowed multiple instances of the same task to be enacted. A possible choice is to select a semantic that satisfies the properties enumerated above from van der Aalst et al’s work [1].

III. How to apply our methodology to declarative-based patterns

Declarative languages allow specification by means of constraints the rules that should be adhered to by the user during workflow execution ("what" to do) while leaving a lot of freedom on the actual workflow that the user chooses to satisfy those constraints ("how" to do it). While CPN-based patterns allow modeling of standard and alternative paths of execution, any of which could be taken depending on some a-priori or run-time available data, they are incapable of handling exceptional or unpredicted situations. In CPN patterns exceptional situations have to be modeled explicitly. However modeling all the possible scenarios can result in very complex models. Even worse, considering that exceptional situations and emergencies can arise at any point in a workflow, CPN-based modeling seems not to be feasible. On the other hand according to [] declarative specifications overcome these problems by allowing more flexible selection of workflows that lead to less complex specifications.

The DECLARE [] language is an example of a declarative formalism which, like CPNs, has a formal unambiguous semantic and a graphical representation. DECLARE provides an inventory of constraint templates and the semantic of each template is given in linear temporal logic.

In Figure 7 a) we give an example of a DECLARE specification containing the following restrictions:
A formal approach to the analysis of CIG languages, Grando et al

1) An exclusive choice between task \( A \) and a trigger: it is not possible that both task \( A \) gets enacted and the trigger is fired.

2) Task \( B \) as a response to task \( A \): whenever task \( A \) is executed activity \( B \) has to be eventually executed afterwards.

3) Task \( B \) as a response to the trigger: whenever a trigger is fired activity \( B \) has to be eventually executed afterwards.

4) Exactly one \( B \): task \( B \) has to be enacted exactly once.

Possible traces of execution for the DECLARE model from figure 7 a) are: \( A.B \) and trigger.B. This model can be used to differentiate control flow in normal and exceptional situations: in the normal flow task \( A \) is followed by task \( B \), but in some cases when an emergency happens a trigger is fired and then task \( A \) is avoided in order to save time and task \( B \) is immediately activated. As in explained in Mulyar et al[] the proposed CPN-based patterns from van der Aalst et al [] do not necessarily reflect all the prototypical control flows required in the specification of workflows. Here we provide additional evidence for their statement by presenting in Figure 7 b) the CPN equivalent to the DECLARE model in Figure 7 a), which is not part of the control flow patterns proposed by van der Aalst et al [1].

An important result from [] is that there is an algorithm to compute from any DECLARE model the state-space graph corresponding to all possible execution traces allowed by the model. Therefore the methodology explained in Section III A) can be used to prove that a workflow-based language satisfies a DECLARE model by: skipping step (2), and replacing step (3) with the execution of the algorithms introduced in [] to obtain the state-space graph from the DECLARE model.

Conclusions and future work

We have mainly used the Colored Petri Net formalism and the notions of congruence and bisimilarity from pi-calculus to explore the benefits of formally analyzing the expressiveness of a
A formal approach to the analysis of CIG languages, Grando et al

workflow-based language. We have exemplified the use of formal strategies for comparing execution flows by proving the satisfaction or non-satisfaction of some of the patterns from van der Aalst et al[1] by the PROforma language. The proofs of non-satisfaction of pattern 5 (Simple Merge) and pattern 8 (Multi-Merge) suggest that the language used in PROforma to express antecedent tasks should be extended to allow the definition of more complex propositional formulas.

The proofs presented here clarify some of the weaknesses of PROforma and could be used to justify and document future revisions of the language.

In Mulyar et al [3] the expressiveness of the clinical workflow languages Asbru [12], GLIF [13], EON [14] and PROforma [15] has been studied with respect to these patterns. For each language considered in Mulyar’s work [3] they checked if it was possible to realize the control flow patterns from [1] with the facilities offered by the language. A pattern was considered to be fully supported if using simple combination of the language constructs it was possible to provide the behavior described by the pattern. A pattern was considered partially supported if complex combinations of pattern constructs (resulting in non intuitive process) could provide the same functionality as the pattern. A pattern was not satisfied if it was not fully or partially satisfied. Their proofs were carried out without applying formal methods of verification, but to overcome the absence of formal strategies they increased the confidence of their results by asking the developers of the four languages to check their results before publication.

In contrast here we have used formal strategies of proof to check the results presented in [3] for the PROforma language, exposing differences between their results and ours. Our study started first with patterns 5, 8 and 10 because initial informal analysis indicated that Mulyar et al’s results[] did not strictly hold. We continued our study with patterns 14 and 15 which consider multiple instances of tasks. Because PROforma has not been designed with enactment of different concurrent and independent instances of the same plan specification in mind, the study of these patterns could suggest how to add this feature to future extensions of PROforma. Finally we studied patterns 23 and 24 because we were interested to explore how different the semantics of the trigger and wait_condition in PROforma were from the standard notion of trigger in other workflow languages. Table 2 summarizes the difference between the results of Mulyar et al [] and ours.
We claim that formal analysis of satisfaction/non satisfaction of patterns can be used to make explicit certain types of strengths and weaknesses of a language. Based on the identified language weakness new extensions can be proposed in a principled way. For instance the proof that PROforma does not satisfy Patterns 5 and 8 induced us to propose an extension to the language that incorporates more expressive notions of antecedent tasks based on the PN formalism, as explained in [27].

We have also indicated how the proof strategies presented here can be easily reused to study the expressiveness of workflow-based languages with respect to patterns of behavior expressed in declarative languages such as DECLARE [] whose formal semantic is expressed in linear temporal logic.

While we have based our study on execution flows, we agree with Peleg et al [18] that other aspects should be also considered for the comparison of clinical workflows languages, for example decision model, data and resources manipulation, flexibility, etc. In the case of the comparative analysis of decision models, logic may be an appropriate framework for comparison studies. But for other aspects of workflow modelling languages different sets of patterns have been proposed, which like van der Aalst et al’s[1] are independent of specific modelling approaches and technologies. For instance patterns for the manipulation of data are presented in Russell et al [23], for resource manipulation patterns are introduced by the same authors in [24] and patterns for flexibility comparisons are explained in [25] [26]. In future work we expect to extend our formal study of workflow-based languages in order to incorporate the mentioned patterns to study workflows expressiveness with respect to data, resources and flexibility.
A formal approach to the analysis of CIG languages, Grando et al.

The health informatics community has recognized the need to enrich the control flow patterns with semantics in order to capture and share good medical practices. For instance in Mulyar et al. work [] they provide a DECLARE specification of a standard diagnosis scenario, or in Grando et al. [] they give a goal-based specification of the typical scenario of diagnosis and treatment of chronic diseases. We think that a future systematic survey of clinical workflows could produce a set of scenario-based patterns that could be of greater interest for the health informatics community. We think that for scenario-based patterns formal proof strategies based on CPNs and model checking could be very appropriate.

While we have focused this paper on the study of language expressiveness we think that the formal strategies presented here could be useful for analyzing workflow interchangeability. In the case of interchangeability an additional problem to consider, besides the bisimilarity of the workflows to be interchanged, is the relationships between the data types (colors) used for the workflow specification. As we explained in Section III.A, two medical workflows $G_1$ and $G_2$ specified in different languages $L_1$ and $L_2$ that conform to the CPN standard could be interchanged if they satisfy an equivalence relationship, even if they are embedded in/connected with other workflows. Based on results from pi-calculus it is guaranteed that the workflows resulting from the interchange will show the same behavior as before the interchange.

We believe that a future promising area of research is the combination of the formal proof strategies presented here with appropriate strategies for data type (ontology) mapping.

As we have seen the notions of congruence and bisimilarity employed here are very useful for the study of the expressiveness of languages. In general the focus during workflow modeling is on the workflow’s goals rather than control flow. Consequently even if two workflows defined in the same language do not have bisimilar control flows, they may still be interchangeable if, based on some formal verification technique (like model checking), they prove to achieve the same goal or intention. This is an interesting area of research that we would like to consider in future work.
Acknowledgements: This work was supported by a programme grant from Cancer Research UK to J. Fox and D. Glasspool.

References

A formal approach to the analysis of CIG languages, Grando et al


A formal approach to the analysis of CIG languages, Grando et al


A formal approach to the analysis of CIG languages, Grando et al


A formal approach to the analysis of CIG languages, Grando et al

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Studied patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Control Patterns</td>
<td>For describing basic aspects of process control such as: sequence, parallel split, synchronization, exclusive choice and simple merge</td>
<td><strong>Pattern 5 (Simple Merge)</strong></td>
</tr>
<tr>
<td>Advanced Branching and Synchronization Patterns</td>
<td>Patterns for analyzing different variants for selecting and continuing the execution of some paths selected from a set of possible paths</td>
<td><strong>Pattern 8 (Multiple Merge)</strong></td>
</tr>
<tr>
<td>Structural Patterns</td>
<td>For determining if the language contains restrictions over process structure</td>
<td><strong>Pattern 10 (Arbitrary cycles)</strong></td>
</tr>
<tr>
<td>Multiple Instance Patterns</td>
<td>Situations where there are multiple threads of execution active in a process model which relate to the same activity</td>
<td><strong>Pattern 14 (Multiple Instances with a priori run-time knowledge), Pattern 15 (Multiple instances without a priori run-time knowledge)</strong></td>
</tr>
<tr>
<td>New Control-Flow Patterns</td>
<td>New patterns and revised variants of patterns from the other categories considering multiple issues like triggers, paths and thread branching and synchronization, and cancellation</td>
<td><strong>Pattern 23 (Transient Trigger), Pattern 24 (Persistent Trigger)</strong></td>
</tr>
</tbody>
</table>

**Table 1:** Summary of the control-flow patterns considered here and proposed by van der Aalst et al in [1]. The patterns in the third column correspond to the patterns on which we have applied the methodology explained here. This paper contains the proof of satisfaction/unsatisfaction in the PROforma language of the patterns in bold type.

<table>
<thead>
<tr>
<th>Patterns</th>
<th>Mulyar et al’s results</th>
<th>Our results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern 5</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Pattern 8</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Pattern 10</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Pattern 14</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
Table 2: Comparison of results obtained by Mulyar et al [3] in their pattern-based analysis of clinical workflow modeling languages, and the results obtained using the formal methodology explained here. The following notation is used: (+) full support, (+/-) partial support, (-) no support.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>15</th>
<th>23</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-</td>
<td>+/</td>
<td>+</td>
</tr>
</tbody>
</table>

Figure 1: PROforma plan described in Section II.A.

PROforma representation adopted through this work: actions as squares, enquiries as diamonds, decisions as circles, plans as squares with rounded corners, keystones as polygons.

Figure 2: CPN describing the workflow of a radiologist. The ovals represent places, the boxes transitions and the labeled arrows the relation between states and transition occurrence. The color associated with each place is indicated below the place in capital letters. The variables \( p \) and \( r \) used for labeling the arcs are respectively of type PATIENT and RADIOLOGIST.

Figure 3: The CPNs drawn as (a) and (b) are trace equivalent but not strong bisimilar; The CPNs drawn as (c) and (d) are weak bisimilar, where \( \lambda \) is an internal task; The CPNs drawn as (c) and (e) are not branching bisimilar.

Figure 4: Pattern 5 (Single Merge) and pattern 8 (Multiple Merge)
(a) Patterns 5 and 8 share the same CPN. Pattern 5 differs from pattern 8 on the restriction that place \( p1 \) can contain at most one token.
(b) PROforma plan \( P1 \) to prove that PROforma does not satisfy patterns 5 and 8 because it is not possible to activate task \( C \) twice. All the components of the plan are keystones.
(c) PROforma plan \( P2 \) to prove that PROforma does not satisfy patterns 5 and 8 because as in (b) it is not possible to merge the flow from the instances of task \( C \) to task \( O \). All the components of the plan are keystones.

Figure 5: Pattern 24 (Persistent trigger)
(a) First pattern variant where triggers are not initiators of task execution and they can be buffered.
A formal approach to the analysis of CIG languages, Grando et al

(b) Second pattern variant where triggers are initiators of task execution
(c) PROforma plan \( P1 \) to prove the satisfaction of the first pattern variant (a). Except for the diamonds that correspond to queries, the rest of the components are keystones.
(d) PROforma plan \( P2 \) to prove the satisfaction of the second pattern variant (b). Except for the diamonds that correspond to queries, the rest of the components are keystones.

**Figure 6:** Proof of satisfaction of pattern 24 (Persistent trigger) in the PROforma language, using the *Colored Petri Net* tool
(a) Second variant of pattern 24
(b) State space graph corresponding to the CPN shown in (a)
(c) CPN equivalent to the PROforma plan shown in Figure 5 (d) for the second variant of pattern 24.
(d) State space graph corresponding to the CPN shown in (c)

**Figure 7:** Proposed pattern for the enactment of tasks in exceptional situations (task \( B \) is normally executed after task \( A \) but in exceptional situations it can be directly triggered without enacting first task \( A \))
(a) DECLARE specification of the proposed pattern
(b) Equivalent CPN representation of the DECLARE model in (a)