Journal of Biomedical Informatics 92 (2019) 103134

Contents lists available at ScienceDirect

Journal of Biomedical Informatics

journal homepage: www.elsevier.com/locate/yjbin

Reasoning about clinical guidelines based on algebraic data types and constraint logic programming

Beatriz Pérez

Department of Mathematics and Computer Science, University of La Rioja, C/ Madre de Dios 53 (Edificio Científico Tecnológico), E-26006 La Rioja, Spain

ARTICLE INFO

Keywords:
Clinical guidelines
Formal verification
Constraint logic programming
Model driven development
Model transformation

ABSTRACT

Previously, the authors presented an overall framework aimed at improving the representation, quality and application of clinical guidelines in daily clinical practice. Regarding the quality improvement of guidelines, we developed a proposal to verify specific requirements in guidelines, using the SPIN *model checker* as verification tool. Additionally, we established a pattern-based approach for defining commonly occurring types of requirements in guidelines, in order to help non experts in their formal specification. In particular, among such patterns, we identified several which could not be verified by using such a proposal, thus leaving their verification as future work.

In this paper, we provide a revised and extended version of that work by providing an overall proposal which mainly addresses previous shortcomings, while providing additional verification functionalities. More specifically, we have defined a complementary proposal to the previous one regarding the verification of guidelines. This proposal uses Formula, a model finding and design space exploration tool that is based on Algebraic Data Types (ADT) and Constraint Logic Programming (CLP). The main contributions of this paper are twofold: (1) providing a more complete set of patterns for defining commonly occurring types of requirements in guidelines, and (2) supporting the verification of a wider range of patterns by combining the use of our previous proposal, based on the SPIN model checker, with our Formula-based method. More specifically, our Formula-based proposal provides us with a solution to the verification of those patterns we were not able to verify previously. Additionally, our proposal has been implemented as an Eclipse plug-in developed based on Model Driven Development (MDD) techniques, which enables us to automatically generate the Formula specification of a guideline, making the process faster and less error-prone than a manual translation. This Formula specification, together with the requirements to be checked in the guideline, are finally taken as input of the Formula tool to check whether the guideline verifies the requirements. We show the feasibility of our overall approach by verifying properties in different clinical guidelines with encouraging results.

1. Introduction

As defined by the Institute of Medicine, clinical guidelines are *systematically developed statements to assist practitioner and patient decisions about appropriate health care for specific clinical circumstances* [1]. They describe the decision points and suitable actions to be carried out depending on a specific patient's state or situation. There are many potential advantages of documenting and using clinical guidelines, among which it is worth noting the assessment and improvement of the quality of care, support for medical decision-making, control of health care costs and reduction of practice variability and the inappropriate use of resources [1,2].

There is a large number of published guidelines since each guideline is focused on a specific condition and on a desired health outcome. Additionally, the adaptations performed in guidelines to be used in concrete hospitals may vary from hospital to hospital since they reflect variations in resources, as well as in the working philosophy of the hospital in question. This fact has motivated to undertake great efforts over the years not only to publish guidelines to make them more accessible, but also to develop computer-interpretable models and tools for the management of guidelines in order to provide guided support to the physician during the application of the guideline. However, the work done on developing, disseminating and computerizing guidelines far exceeds the efforts in improving their quality [3,4]. The fact is that although they are developed by collaboration and consensus among experts taking into account evidence-based medicine and daily medical practice, this process has limitations and can lead to flawed conclusions [5]. In addition, guidelines are commonly represented in natural

E-mail address: beatriz.perez@unirioja.es.

https://doi.org/10.1016/j.jbi.2019.103134
Received 17 August 2018; Received in revised form 17 February 2019; Accepted 18 February 2019
Available online 01 March 2019
1532-0464/ © 2019 Elsevier Inc. All rights reserved.

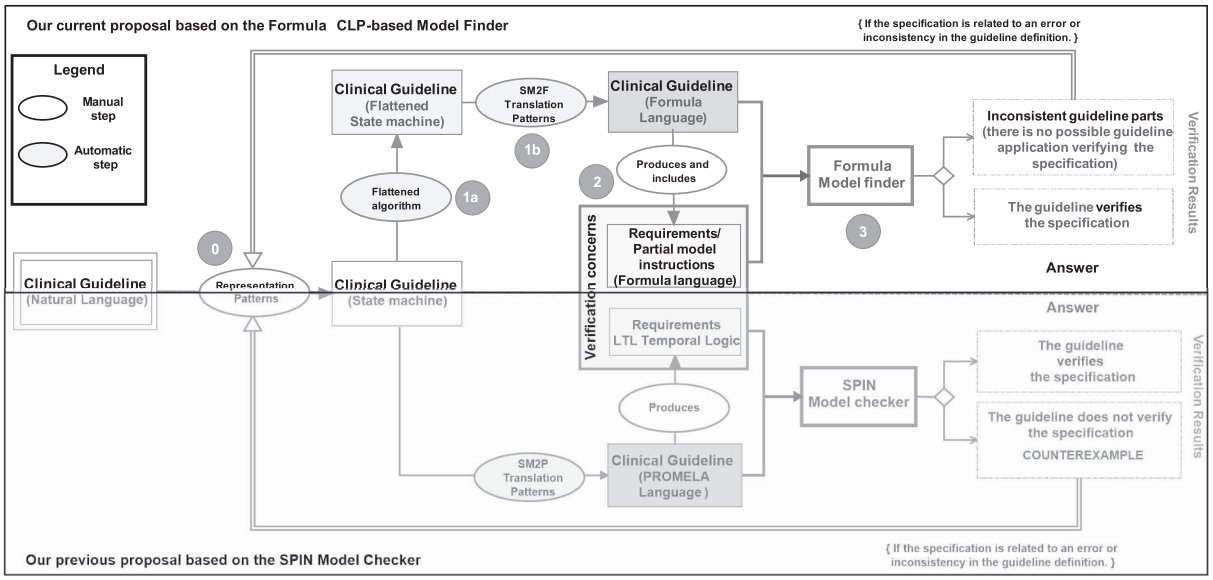

Fig. 1. Overall proposed workflow.

language, which makes them accessible to practitioners, but they can also contain ambiguities possibly leading to their inappropriate use. As a consequence, most clinical guidelines are lacking in quality because of the inconsistency and poverty of the methodological rigor used to define them.

Previously, the author of this paper together with I. Porres [6] presented an overall framework aimed at improving the representation, quality and application of clinical guidelines in daily clinical practice. More specifically, the result presented in [6] was threefold. Firstly, we proposed an approach to represent clinical guidelines using UML State Machines [7] as visual computer language. Secondly, we developed a framework to verify specific requirements in guidelines, using the SPIN model checker [8] as verification tool, in order to check guidelines against semantic errors and inconsistencies in their definition. Additionally, as part of this second contribution, we established a pattern-based approach for defining commonly occurring types of requirements in guidelines in order to help non experts in their formal specification. In particular, among the set of patterns we provided in [6], we identified several which could not be represented in the SPIN specification language, thus leaving the verification of such types of properties as future work. Finally, we presented a framework to develop computer-assisted tools for the application of guidelines previously checked against error or inconsistencies in their definition.

In this paper, we provide a revised and extended version of the work presented in [6], focusing on the quality improvement of clinical guidelines. More specifically, we complete our previous work by providing an overall proposal which aims at verifying clinical guidelines by using a complementary formal method. This formal method mainly addresses the shortcomings of our previous approach, while providing additional verification functionalities. Briefly speaking, first, starting from the set of patterns we established previously for representing commonly occurring types of requirements, we revise and extend them, providing a more complete set of patterns. Second, we develop a complementary proposal for the verification of guidelines using a formal method different than the SPIN model checker. In particular, we propose to use a model finder based on Algebraic Data Types (ADT) and Constraint Logic Programming (CLP) [9] called *Formula* [10]. Formula provides us with a solution to the verification of the patterns we were not able to verify with our proposal based on the SPIN model checker. Overall, the work presented in this paper constitutes a revised and extended version of the proposal we gave in [6] with the main contribution being twofold: (1) providing a more complete set of patterns for defining commonly occurring types of requirements in guidelines in order to help non experts in their formal specification, and (2) supporting the verification of a wider range of patterns by combining the use of our previous proposal based on the SPIN model checker together with our Formula-based method.

The paper is structured as follows. Next section outlines the background and the motivation of this work, based on the representation and quality improvement of clinical guidelines we proposed in [6]. In Section 3, we present the extension of the pattern-based approach we gave for defining commonly occurring types of requirements. In Section 4 we give a brief overview of our proposal based on the model finder *Formula*, while in Section 5 we briefly describe the Formula language. A detailed explanation of the steps involved in our current proposal is presented in Sections 6 and 7. Our experience with applying our approach to a specific guideline used as case study is presented in Section 8. Section 9 discusses the strengths and weaknesses of our approach, while Section 10 presents related work. Finally, conclusions are set out in Section 11.

2. Background and case study

To establish the basis on which our proposal is made, in this section we outline general background information of the framework we presented in [6], focusing on aspects related to the representation and quality improvement of guidelines. In Fig. 1 we depict an overview of our overall framework, which includes both the previous framework presented in [6] (see bottom of Fig. 1), and its extension presented in this paper (see top of Fig. 1). To facilitate the understandability of our proposal, we will use a specific clinical guideline as case study throughout the paper.

Case study. We have chosen a laboratory guideline used to carry out the aliquoting process which can be used in the daily laboratory routine (from now on we refer to this guideline as *AP Guideline*). In particular, this guideline was set within the “Aragon Workers Health Study” (AWHS) project, which has been carried out within a framework of longitudinal management of the data related to the storage of biological samples in a biobank [11,12]. We have chosen this example to illustrate our proposal for its simplicity in respect to other more complex guidelines.

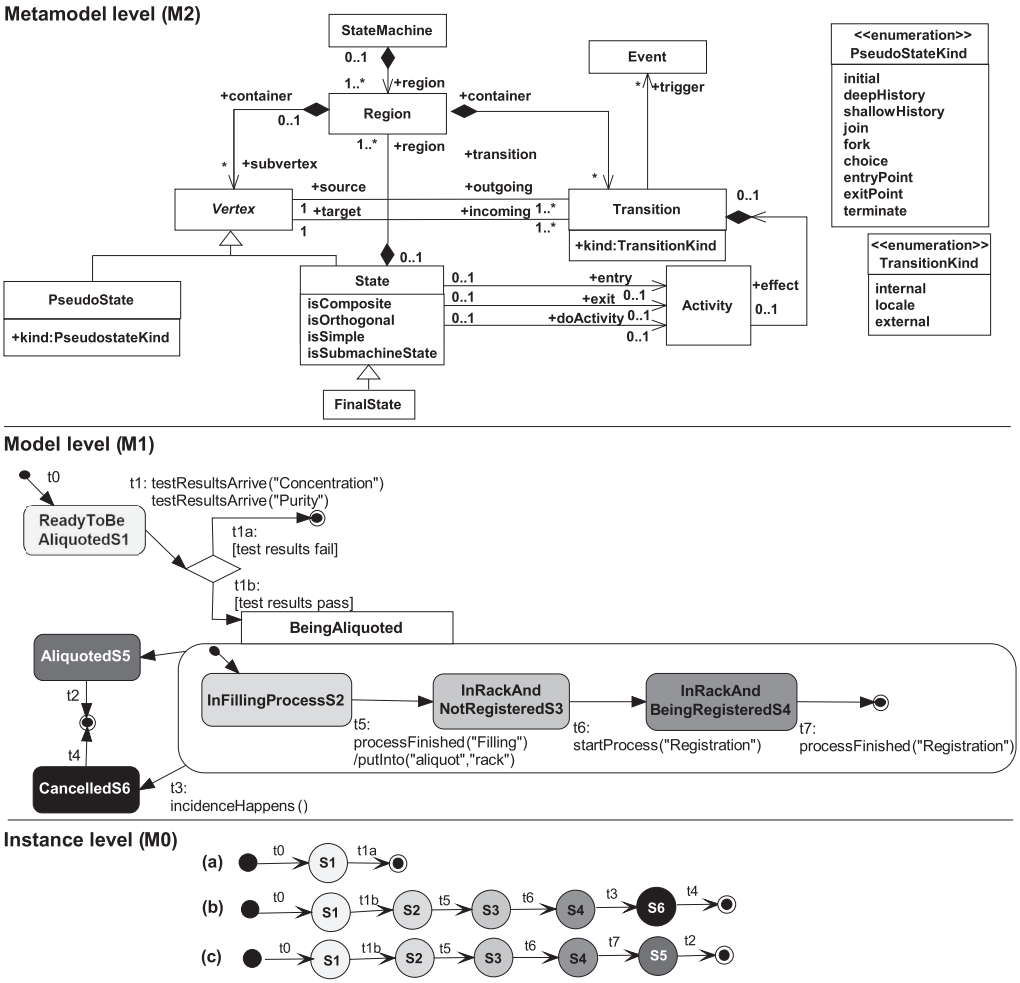

Fig. 2. MOF model levels concerning UML State Machines applied to our case study.

2.1. Our previous work on improving the representation of clinical guidelines

Aimed at improving the representation of clinical guidelines, previously we have proposed together with Porres [6] and Domínguez et al. [13] the use of UML State Machines [7] as a method for representing the dynamics of guidelines. We also provided several patterns to assist in the modeling process to have a better and more understandable representation of every guideline (see center left side of Fig. 1). These patterns took into account the specific elements and semantics of UML State Machines and provided representation rules of guidelines by using UML in the medical context. To facilitate the understanding of our overall proposal, next we rely on Fig. 2 to give a brief overview of both UML State Machines background within the Meta-Object Facility (MOF) framework, and our patterns for representing guidelines as state machines. In particular, MOF is a metadata management framework which enables development and interoperability of model and metadata driven systems. It originates from the context of model-driven engineering [14,15] and was originally built in response to a need for providing a metamodeling architecture to define UML [16]. Among the four levels identified by MOF, we note the first three which are denoted by convention as M0, M1 and M2 (from a minor to major abstraction). Fig. 2 presents such three levels concerning UML State Machines, applied to our case study: the *Metamodel level (M2)*, with an excerpt of the UML State Machine metamodel (see top side), the *Model level (M1)*, with the concrete state machine for our case study (see center side), and the *Instance level (M0)*, with three different sequences of states an aliquot object can go through during its lifetime (see bottom side).

Metamodel level. A state machine consists essentially of *states*, *transitions* and various other types of vertexes named *pseudostates* [7]. Firstly, *states* denote a situation of objects during which some condition holds. There are three kinds of states distinguishing among *simple*, *composite* or *submachine* states. *Simple* states are characterized by not having nested states (substates), while *composite* states are divided into *orthogonal composite states*, used to model concurrent behaviors where several states are active simultaneously, and *simple composite states*, used to specify that only one of their substates must be active at a same time. A *submachine* state is a special kind of a *state* that refers to another defined state machine diagram so that it can be reused. *Composite* states can have one or more *regions* which are considered as simple containers of a connected set of substates, pseudostates and transitions. Consequently, a particular "state" an object may be in at a given time can be represented by one or more hierarchies of states, starting with the topmost regions and down through the composition hierarchy to the simple states. This complex hierarchy of states is referred to as a *state configuration* [7]. A *transition* is the mechanism by means of which an object leaves a state configuration and changes to a new state configuration. Such a change can be triggered by some *event*. Particularly, a transition is a directed relationship between a source vertex and a target vertex, where these vertexes can be either *states* or *pseudostates*. A *pseudostate* is an abstraction used to connect multiple transitions into more complex state transitions paths. There are several kinds of pseudostates, such as *initial* or *choice* (see top side of Fig. 2). It is worth noting the subtle difference between *terminate* pseudostate (shown as a cross) and *final state* (shown as a circle surrounding a small solid filled

circle). On one hand, a *final state* is a special kind of state that represents the completion of the enclosing region. Thus, if the enclosing region is directly contained in a state machine (it is a topmost region) and all other regions in the state machine also are completed, then it means that the entire state machine is completed. On the other hand, entering a *terminate* pseudostate implies that the execution of the state machine is terminated immediately (it is equivalent to invoking a destroy object action).

Model level. Taken this general background information into account, in [6,13] we defined our representation patterns so that we can specify a clinical guideline by means of a UML state machine conforming to the UML State Machine Metamodel. So that the reader can get an intuitive understanding of our patterns, on the center side of Fig. 2 we present the state machine resulting from the application of such patterns to the *AP guideline*. More specifically, this state machine represents the fact that over the course of the life of an aliquot, it can mainly take up four states: *ReadyToBeAliquotedS1*, *BeingAliquoted*, *AliquotedS5* and *CancelledS6*. The *initial* pseudostate, depicted as a filled circle, represents the creation of the aliquot. The starting point itself of the guideline corresponds to the simple state in which the aliquot is ready to be aliquoted (simple state *ReadyToBeAliquotedS1*). If the results of the concentration of the sample and the purity tests are ready (event *testResultsArrive*), two options are distinguished depending on whether such results are *failed* or *passed* (represented by a *choice* pseudostate). If *failed*, the process leads to a *final state* which means that the guideline application finishes. Otherwise, the aliquoting process starts and the aliquot changes its state from *ReadyToBeAliquotedS1* to that of *BeingAliquoted* (which corresponds to a *simple composite state*). In this state, the aliquot is filled (simple state *InFillingProcessS2*) and when it finishes, the aliquot is put into the corresponding rack, changing its state to *InRackAndNotRegisteredS3*. When the process of registration starts (event *startProcess("Registration")*), the aliquot's state changes to *InRackAndBeingRegisteredS4*. When such a process finishes and no incidence has taken place, the aliquot process is completed (state *AliquotedS5*), which finally leads to a *final state* as previously. If during the application of the guideline an incidence takes place (event *incidenceHappens()*), the aliquoting process of such an aliquot is cancelled (state *CancelledS6*), and the process leads to the *final state*, as in the previous case.

Instance level. At this level, the sequence of state configurations an object can go through during its lifetime is represented, which is known as *execution traces*. We note that an object can only be in exactly one state configuration at a time, which is referred to as its *active state configuration* [7]. On the bottom side of Fig. 2 we show three of the possible *execution traces* of an aliquot.

2.2. Our previous work on improving the quality of clinical guidelines

Before developing the computer-assisted tool for the application of a guideline, we carried out a preceding step [6], which consisted of performing a formal verification of the guideline so that the corresponding computer-assisted tool was generated from the guideline with lack of errors or inconsistencies in its definition. Such a verification process aimed at checking whether a given guideline satisfied or not a number of desired requirements. At this point, two issues came into play. On one hand, such requirements had to be established and formally defined. On the other hand, we had to use a formal verification technique which provided us with an effective and efficient way to verify required properties in a guideline. Next, we tackle these two aspects in more detail.

2.2.1. Specification of the requirements to be verified in clinical guidelines

We took into account two main aspects. First, formal verification processes require properties to be specified using a mathematical formalism (such as temporal logics). Since practitioners usually do not have solid mathematical backgrounds [17–19], this step constitutes a big challenge since it requires bridging the gap between the natural language in which the properties are elicited from domain experts and the rigorous, but usually not trivial to specify correctly, mathematical formalism. Second, the accurate representation of properties can also constitute a surprisingly difficult task because of all the details that must be taken into account [17]. For this reason, in [6] we established a hierarchy of property specification patterns for defining commonly occurring types of requirements in guidelines (whenever in the following text the term "pattern" without any qualifier appears, it refers to a property specification pattern). These patterns, which were drawn from a thoroughly literature survey in different domains (not limited just to the medical domain), aimed at enabling non-experts in the specification language of the tool, to easily write formal specifications, thus easing the verification process. The idea around such patterns is that a non-expert formulates in natural language the property she/he wants to be verified in a guideline. Such a property would conform to one of the defined patterns which, by providing the mapping of the property to formal specification languages, enables her/him to easily formulate the property in the formal language, so that it can be checked in the guideline. In Appendix A we give several guidelines and recommendations to use property specification patterns in general, and our overall hierarchy of patterns, in particular.

In order to make sure that our set of patterns was complete enough for representing the widest possible spectrum of guideline properties, we performed a thorough review of works which dealt with the formal proving of guidelines. In such review we both (1) collected the properties these works considered useful to be verified in guidelines, and (2) identified whether such properties matched or not any of our patterns. The results obtained from this analysis contributed to establish our final property specification pattern hierarchy, which is shown in Fig. 3. More specifically, our hierarchy of patterns was built upon the patterns given by Dwyer et al.'s [18] and by Ryndina et al.'s [20,21]. On one hand, Dwyer et al. established the hierarchy of patterns shown as dark grey squares in Fig. 3, where each property specification pattern consisted of a *pattern* and a *scope*. *Patterns* are classified into *occurrence* and *order* patterns, and specify *what* must occur. A *scope* defines a starting and an ending state/event for a pattern, distinguishing among *Globally*, *Before*, *After*, *Between…And*, and *After…Until*. For each specification pattern, Dwyer et al. provided mappings to several formal specification languages (such as Linear Temporal Logic (LTL) [22] or Computational Tree Logic (CTL) [23,24]) presented as temporal logic formulas. On the other hand, Ryndina et al.'s patterns particularly constituted an extension of Dwyer et al.'s *Existence* pattern, considering four new categories of such a pattern (see light grey squares in Fig. 3). We particularly considered this extension of Dwyer et al.'s patterns given the presence of properties of *existential nature*[1] within the properties we collected from the literature. The set of properties gathered from the literature led us also to include another 4 subpatterns within the *Existence* pattern (see white squares in Fig. 3), thus obtaining our final hierarchy of patterns. For a more complete explanation of this hierarchy of patterns, we refer the reader to [6]. Because of their relevance to the work presented in this paper, next, we provide a brief introduction to both LTL and CTL (for more detail about LTL and CTL, see [22–24], respectively).

> *LTL.* An LTL formula is built up from a set of atomic proposition variables $p,q,\ldots$, the usual logic connectives: $\neg$, $\vee$, $\wedge$, $\rightarrow$, and the following temporal modal operators: `X` for next, `G` for always (globally), `F` for eventually (in the future), `U` for until, `R` for release. In LTL, the truth of a temporal logic formula is defined on a path, that is, propositions are interpreted not over trees (like CTL does) but over individual paths [22]. A system satisfies a formula if all

[1] Related to the existence of at least one path in the guideline application in which some condition must hold.

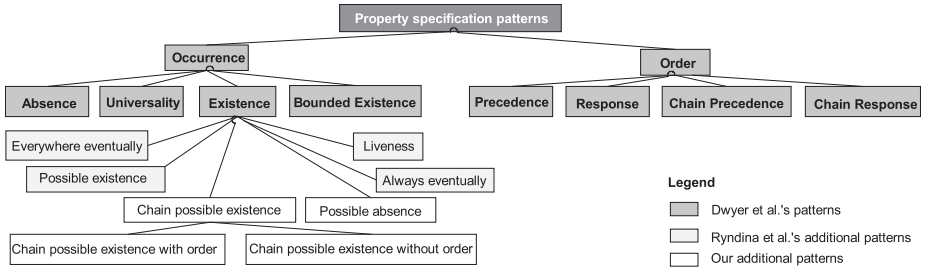

Fig. 3. Our Property specification pattern hierarchy proposed in [6].

Table 1
Several LTL temporal formulas.

Operator	Explanation	Diagram examples
Xp	This formula holds iff p holds at the next state of the path.	
Gp	This formula holds iff p holds globally, that is, p holds at every state along the path.	
Fp	This formula holds iff p eventually holds, that is, if p holds at some state along the path (somewhere on the subsequent path).	
pUq	This formula holds iff q holds at the current or a future position, and p has to hold until that position. At that position p does not have to hold any more (that is, this formula holds, if p holds until q occurs).	
pRq	This formula holds (p releases q) iff q is true until the first position in which p is true. If p never becomes true, q must remain true forever.	
GFp	This formula holds iff p is true infinitely often throughout the execution of the system, that is, at any instant of the execution of the system you can always find a later instant so that p is true.	
FGp	This formula holds iff for every execution of the system, there is a moment from which on p holds forever, that is, at some instant in the future p will stay true for the remaining of the execution.	

execution paths satisfy the formula (LTL assumes implicit *universal quantification* over paths). Particularly, a LTL formula p is satisfied in a path x (depicted as $x \vDash p$) iff p is satisfied for a starting position on that path x. Table 1 shows basic LTL formulas and examples, including also FGp and GFp which will be cited later.

CTL. It is a *branching-time logic*, that is, the computation starting from a state is viewed as a tree where each branch corresponds to a path [23,24]. Thus, a state now may have more than one successor [25]. In contrast to LTL, the truth of a CTL formula is defined on states (so all CTL formulas are called "state formulas"). A formula in CTL is built from atomic proposition variables *p,q,…*, the usual logic connectives (as in LTL), path quantifiers (A for "for all paths", and E for "there exists a path"), and LTL-like *temporal operators* (X, G, F or U). Particularly, CTL temporal operators are generated from the basic temporal operators X, G, F and U, combined with one of the path quantifiers A or E as a prefix, ending up with eight temporal operators distinguishing between *universal modalities* (AX, AG, AF and AU) and *existential modalities* (EX, EG, EF, and EU). So, given a *path* formula φ (that is, Xp, Gp, Fp or pUq), Aφ and Eφ will be *state* formulas with the following interpretation: Aφ is true on a state iff φ is true for all paths originating from that state. Eφ is true on a state iff there exists a path originating from that state where φ holds (see Fig. 4 for a description of basic CTL temporal formulas). Thus, CTL allows explicit *existential* and *universal* quantification over all paths.

2.2.2. Verification process

As for the second issue, a wide number of efforts have been made to stimulate the improvement of clinical guidelines [25–36] using different verification techniques including, for example, model checking, theorem proving, or knowledge-based [6]. These techniques basically follow the same outline of a verification process. First, a guideline is modeled in some predefined guideline representation language (GLIF, PROforma, Asbru, GLARE, etc.) obtaining a first model. Second, this model is translated into the input specification language of the chosen verification technique (such as PROMELA or PROcess MEta LAnguage [8]), resulting in a second model. Third, the desired properties are specified in a formal language (such as LTL, CTL, ACTL (Action Computation Tree Logic), or a variant of ITL (Interval Temporal Logic) [37]). Finally, the properties are checked in this second model to verify whether the guideline satisfies them or not.

In [6] we decided to use a model checking technique, in particular, the SPIN [8] model checker which uses PROMELA, as the input specification language, and LTL formulas, to specify the properties to be verified in the model. In general terms, *model checking* represents a

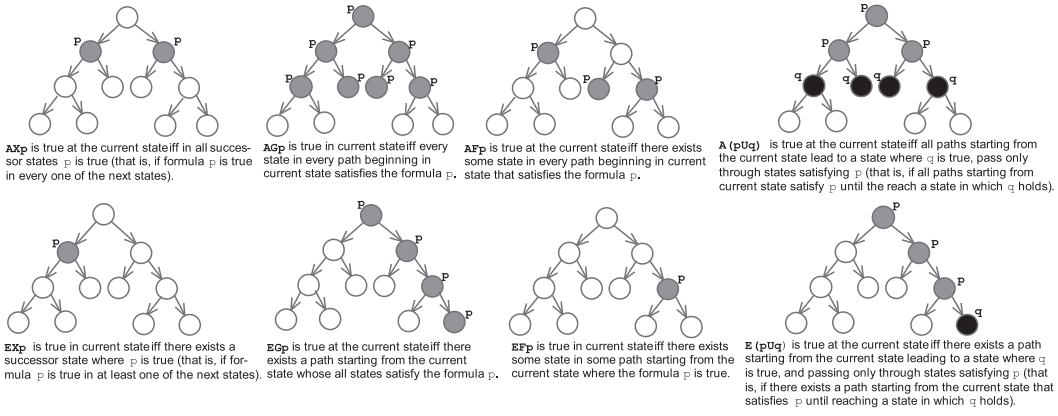

Fig. 4. Several CTL temporal formulas.

Table 2
Number of properties matching each pattern.

Our patterns in [6]			References					
			Rutle et al. [26]	Hommerson et al. [25]	Rahman and Bowles [28]	Terenziani, Giordano et al. [27]	Simalatsa et al. [29]	Kamsu-Foguem et al. [30]
Occurrence	Universal	Global	1			2		
		After						
	Absence (basic)	Global			2			
		Before						
		After				3		
	Existence	Global				2		
	Possible	Global		1		4		
	Existence	After				5	1	
	Possible	Global			2	3		
	absence	After			1	2		
Order	Response (eventual)	Global		2		8		3

formal technique for verifying whether a system satisfies a specification or formula p [38] (as an example, the verification process regarding the use of the SPIN model checker can be seen on the bottom right side of Fig. 1). The entries of a model checker are both a model of the system under verification (in our particular case, the guideline specified in PROMELA) and a property the model should meet. In case the system meets the property, a model checker will return a message informing the user about the fact. However, if any behavior of the system under verification violates the property, the negative answer to the verification is supported with a so called *counterexample*, that is, a behavior of the system which witnesses the invalidity of the property. Regarding LTL [22], as described previously, it assumes implicit *universal quantification* over paths. For this reason, properties which assert the existence of a path cannot be expressed in LTL. However, in these cases in which the property to verify does not mix universal and existential path quantifiers, LTL can be used for proving the negation of the property, interpreting the result accordingly [39]. It suffices to submit the negation of the property for verification (that is, the formula $\neg p$). If there exists a counterexample for $\neg p$, then such a trace must necessarily satisfy p, giving by the same occasion a valid deployment sequence.

Based on the SPIN model checking process, in [6] we obtain the PROMELA model representing the guideline under verification by, first, applying our guideline to state machine representation patterns to obtain the guideline's state machine, and second, translating such a state machine into PROMELA (see *SM2P translation patterns* on the bottom center side of Fig. 1). The properties to be verified in the guideline are manually specified in LTL by using our property specification patterns [6]. We supported the translation of the guideline's state machine to PROMELA by using a MDD-based approach which automatically processed the state machine and generated the PROMELA model.

While our previous proposal was proven to provide encouraging results for guidelines' verification, we recognized certain limitations [6]. More specifically, we have considered it necessary to complete and improve such a work in two different lines. First, we aim at revising our previous patterns to include new properties recent works consider useful to be verified in guidelines. The second line is mainly related to the verification of concrete properties considered in our previous hierarchy of patterns which we could not verify with our previous framework, in particular, properties conforming to the *Liveness* pattern. This pattern states that "at any time during the execution of the system, something will eventually become possible". This pattern is not supported by LTL [40,41]; neither its formula nor its negation are represented in LTL. As a consequence, as we stated in [6], model checking techniques could not be used to decide whether this kind of formula is true or not [38]. This issue made us postpone the verification of such types of properties as part of future work. So, in this paper we present an approach based on the Formula model finder which allows us to verify such properties.

3. Extending our property specification patterns

Aimed at revising our previous work in [6] regarding our Property specification patterns, we have analyzed different works which have tackled the formal verification of clinical guidelines and which have been published in the literature after our proposal in [6]. Additionally, we have looked at the literature on Property specification patterns in general, aimed at revising our previous patterns. Such analysis led us to extending the set of patterns by (1) specializing several of our patterns by providing a more exhaustive subclassification, and (2) defining new patterns.

More specifically, stating from the works tackling guidelines' verification, we have followed a documentation process similar to the one we used in [6]. First, we have identified the properties the analyzed works consider useful to be verified in guidelines, and we have abstracted them from particularities. Based on the collected properties, we have started from our Property specification patterns in [6] and we have manually decided whether each Property matches any of our previously defined patterns. In particular, based on [18] (pp. 416), for each Property we have recorded, when possible, the following information: (1) the description of the Property in natural language (*Requirement*), (2) the pattern to which the Property belongs (*Pattern*), (3) the scope of the pattern (*Scope*), (4) the parameters provided to the pattern (*Parameters*), (5) the formal specification of the Property in the formal specification language chosen by the authors (CTL, LTL, etc.), (6) the source of the property, such as the authors and citation of the paper (*Source*), (7) the specific clinical guideline for which the Property is defined (*Context*) and (8) any additional information needed (*Note*). Performing the documentation process has required significant effort since it has been necessary to thoroughly read each paper, understand the description in natural language of the clinical guideline used (in several cases the guideline was not entirely provided), and identify the formal specification of the properties given in the paper. As a result, we have collected 45 requirements from the analyzed papers. Due to space reasons, we do not include the complete documentation of these properties, but we show in Table 2 the classification of the properties defined by the most significant analyzed papers. In particular, from this analysis we have concluded that, while the vast majority of the properties match one or other of our previously defined patterns, there are some of them which do not (see Fig. 5). More specifically, the selected properties which do not match any of our previous patterns, together with the analysis of the literature on Property specification patterns in general, have led to inclusion of the new patterns and subpatterns presented in this paper. Next, we explain in detail the revision we have made to our previous hierarchy of specification patterns presented in

Requirement: During the cancer treatment the patient should follow either *ProcedureA* or *ProcedureB*, repeatedly.
Pattern: Unknown
Scope: Unknown
Parameters:
LTL: GF(ProcedureA.running || ProcedureB.running)
Source: A. Rutle et al. [26](pp. 323)
Context: Cancer treatment
Note: The specification of the requirement is not explicitly provided in the paper.

(a)

Requirement: Eventually normal blood pressure (*normotension*) will always hold.
Pattern: Unknown
Scope: Unknown
Parameters: Propositional
LTL:FG normotension
Source: A. Hommersom et al.[25](pp.70)
Context: A guideline that deals with breast cancer.

(b)

Requirement: There exists some state after which all patient groups have normotension at the same time.
Pattern: Unknown
Scope: Unknown
Parameters: Propositional
CTL: AF(AG normotension)
Source: A. Hommersom et al.[25](pp.70)
Context: A guideline that deals with breast cancer.

(c)

Fig. 5. Properties which do not match with our previous patterns.

[6]. We also included in our explanations examples of application of most of the proposed new patterns (in addition to those presented in Fig. 5).

3.1. Occurrence patterns

Based on the revised works tackling the formal verification of guidelines, we have decided to extend our patterns included in this category in the following way (see Table 3).

Fairness pattern. Among the properties identified in Fig. 5, Property labeled (a) not only refers to the fact that a specific process is possible during the application of the guideline, but it also requires that such a process must be possible consistently throughout the guideline's execution. This type of Property is called *Fairness* [42], in particular, *Unconditional fairness*. Properties describing these requirements cannot be expressed directly in CTL [43]. Among the several types of fairness in the literature (for a comprehensive study of the matter see [44]), we have considered not only the one represented by Property (a) (*Unconditional fairness*) but also another two for being the most used within a wide number of contexts [44,45] (*Weak* and *Strong fairness*). In Table 3 we present the explanation of these patterns. As *example of application*, we note that *Weak* and *Strong fairness* properties can be defined in terms of processes/resources that can be enabled/available and finally executed/used, respectively. Let's suppose that a specific test is performed by a concrete medical equipment, and a concrete guideline recommends such a test, so it depends on the availability of such an equipment. If p refers to the positive availability of such a test (the equipment is available), and q corresponds to finally performing the test, we could be interested in checking that "if the test is almost always available, it should be executed infinitely often" (*Weak*), or "if the test is infinitely often enabled, then it should be performed almost always (*Strong*).

Persistence pattern. Regarding properties labeled (b) and (c) (see Fig. 5), we note that precisely they correspond to two well-known "somehow" related formulas in LTL and CTL (see Table 3). As stated in [46], the satisfaction of $AF(AGp)$ forces all paths to have a state s such that all paths starting from s always satisfy p. In contrast to that, FGp is satisfied if on all paths a state s is reached such that all the following states on the path satisfy p, but there might be other paths starting from s which reach states satisfying $\neg p$. Aimed at explaining the subtle difference between these two formulas, on top of Fig. 6 we show an example obtained from [47] of a structure where FGp holds but $AF(AGp)$ fails. The reason is that the LTL Property is related to the individual paths, and on any infinite path of the given structure we can reach a state a from which p will holds forever. On the other hand, the CTL formula $AF(AGp)$ requires that on all paths from the state a, we can reach a state satisfying AGp. Note that the only state satisfying AGp is the state c; however, the structure does not satisfy $AF\{c\}$-as shown in the right hand side of the figure, the left most path of the computation tree is a counterexample. On this particular path, we can stay in the state a while reserving the possibility of going to the state b (where p does not hold) [47]. Taking this into account, although slightly different in semantics, we have considered these two formulas as part of the *persistence* pattern (see Table 3). At this point, we would like to note the difference between $AF(AGp)$ and the *Always eventually* pattern presented in [6]. This latter pattern, which corresponds to a subpattern of the *Existence* pattern (see Fig. 3), is represented in CTL as $AG(AFp)$, stating that "from any reachable state, a state where p is asserted must be reached".

On the other hand, based on the revised works tackling Property patterns in general, we have decided to extend our previous proposal by specializing the subpatterns *Absence* and *Liveness*. In particular, while *absence* (or *safety*) properties are normally used to represent that something bad never happens, *liveness* properties are commonly used to verify that something good happens.

Absence pattern. In [6] we adopted the *Absence* pattern from Dwyer et al.'s patterns [18] to verify that the system execution is free of certain events or states. The global scope of this pattern is represented as $AG(\neg p)$, in CTL, and $G(\neg p)$, in LTL. This formula is commonly used to assert that "nothing bad should happen", that is, that "a bad behavior should never occur" [48]. We have identified three specializations of this pattern for being broadly used [48,49]: *Basic absence*, *Mutual*

Table 3
New occurrence patterns.

Fairness pattern		
	Unconditional fairness	Also known as *impartiality*, is used to express that some event or state (p) is infinitely often throughout the execution of the system [42,43]. **CTL:** Not supported. **LTL:** GFp
	Weak fairness	Also called *justice* [45], asserts that if an event or state (p) is true almost everywhere (is continuously true), then another event or state (q) will occur/be true infinitely often. **CTL:** Not supported. **LTL:** FGp → GFq [44]
	Strong fairness	Also called *compassion* [45], asserts that if an event or state (p) is true infinitely often (that is, repeatedly), then another event or state (q) will occur/be true almost everywhere. **CTL:** Not supported. **LTL:** GFp → FGq [44]
Persistence pattern		This pattern follows the semantics and considerations described in the text. More specifically, AF(AGp) is used to express that on all execution of the system, there is a moment when it is guaranteed that from now on p holds forever. FGp is true if for every execution of the system, there is a moment from which on p holds forever. The LTL version is referred to as *Eventually always* [44]. **CTL:** AF(AGp) **LTL:**FGp
Absence pattern		
	Basic absence	Defined to represent the basic representation of the *Absence pattern* (***p*** is false), used to represent the fact that the system is free of a certain event or state p. **CTL:** AG(¬p) **LTL:** G(¬p)
	Mutual exclusion	The system execution never reaches a situation where two events or states (p and q) are in the critical section at the same time. **CTL:** AG(¬(p∧q)) **LTL:** G(¬(p∧q))
	Partial correctness	It can be interpreted as saying that the system execution does not produce the wrong answer. As stated in [49], partial correctness may be specified in terms of a precondition p, which must hold initially, a postcondition q, which must hold on termination, and a condition a, which indicates when termination has been reached. **CTL:** p → AG(a → q) **LTL:** p → G(a → q)
Liveness pattern		
	Reachability	To represent our basic formulation of the*Liveness* pattern. More specifically, "at any time during the execution of the system, something (p) will eventually become possible" [6]. This Property is also known as possibility [42], *(nested) reachability*. **CTL:** AG(EFp) **LTL:** Not supported
	Temporal implication	For all states (AG) where some condition p holds, there exists an execution path where a state will eventually be reached (EF) in which the condition q holds [42]. More specifically, whenever p happens, eventually q will happen. **CTL:** AG(p → EFq) **LTL:** Not supported

exclusion, and *Partial correctness* (see Table 3). In the particular case of the *Basic absence*, it corresponds to our previous *Absence* pattern. We consider *Basic absence* properties as safety properties since they imply that certain undesirable states are unreachable. This is the case of properties representing *deadlock freedom*, which aim to prove that the system never reaches a state where no moves are possible (i.e., the system does not reach a *deadlock* state). As *examples*, we note [28], where authors illustrate the verification of *Basic absence* properties in the context of an algorithm for blood glucose lowering therapy in adults with type 2 diabetes. In particular, authors define a Property to check whether the guideline includes potential *deadlocks*. Also, authors define the Property AG (¬ diab.SecondIntensification) (the therapy includes two intensifications), which holds iff second intensification is never reached (*universal quantification*). We also want to note that, as stated in [28], the fact that this formula is not satisfied will result in a trace that shows a second intensification of the therapy being reached (*existential quantification*). Additionally, a natural use of the *Mutual exclusion* pattern would be to check whether a guideline suggests

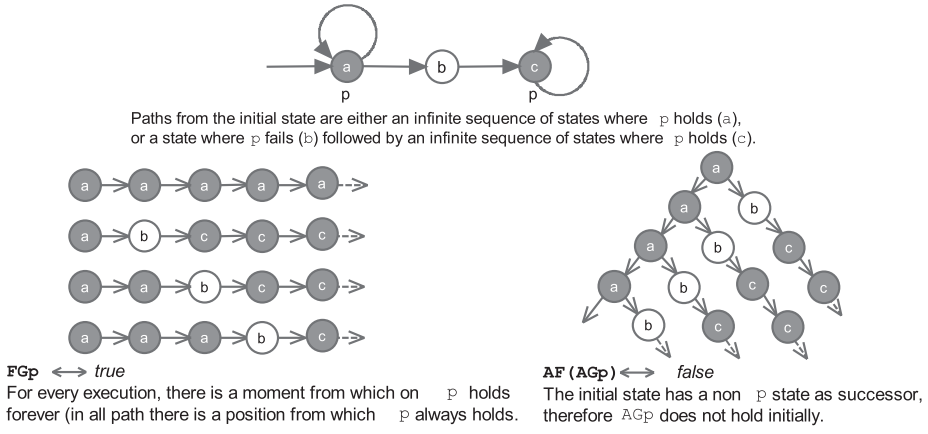

Fig. 6. Example of AF(AGp) and *FGp* formulas.

Table 4
New order patterns.

Response pattern		
	Eventual response	To express that some event or state p will always lead, at some point in the future, to another event or state q. **CTL:** AG (p → AF q) **LTL:** G (p → F q)
	Immediate response	To express that some event or state p will always immediately lead to another event or state q. **CTL:** AG (p → AX q) **LTL:** G (p → X q)
Constrained chain		If a state/event p occurs in a path, then states/events q1 and q2 occur in that order, and the state/event q3 shall not occur. **CTL:** AG(p → AF(q1 ∧¬q3 ∧AX(A(¬q3 U q2)))) **LTL:** G(p → F(q1 ∧¬q3 ∧X(¬q3 U q2)))

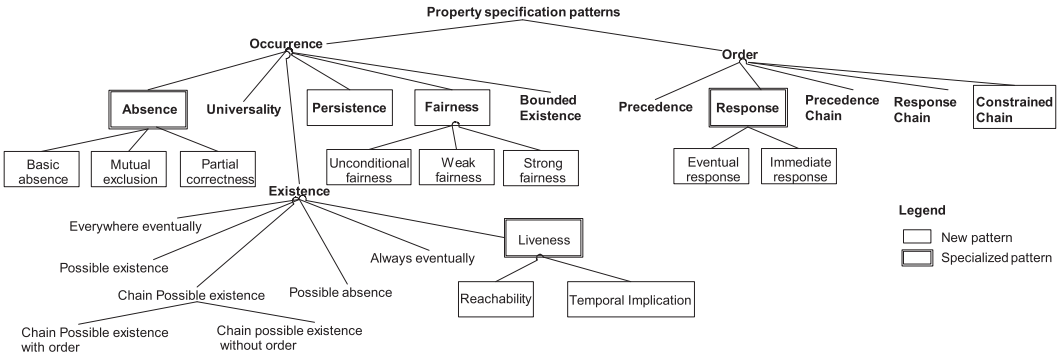

Fig. 7. Our extended Property specification pattern hierarchy.

applying to the patient two mutually exclusive treatments at the same time.

Liveness pattern. The *Liveness* pattern was included in [6] to verify that "at any time during the execution of a system, something will eventually become possible". This pattern is normally used to assert that "something good eventually happens", or more specifically, that a Property that requires desired events, eventually occurs. We have specialized this pattern by considering two specializations: *Reachability*, which represents our basic formulation of the *Liveness* pattern, and *Temporal implication*, obtained from [42] (see Table 3). As *example of application*, in [50], the author describes the use of the *Temporal implication* pattern to check that "if the user chooses to undergo *a concrete* surgery, then the outcomes of this must eventually be presented". An example of the *Reachability* pattern applied to our case study will be presented in Section 7.

3.2. Order patterns

The new patterns of this category, which have been included based on the literature tackling specification patterns in general, are presented in Table 4.

Response pattern. Taking into account the properties extracted from the papers tackling the formal verification of clinical guidelines, we have noted that there are a non-negligible number of properties which match the *Response* pattern. In [6], we adopted this pattern from Dwyer et al. to represent causal relations between two states or events and, in particular, one state/event leading to another. The *Response* pattern in Dwyer et al. is represented by AG(p → AFq), in CTL, and G(p → Fq), in LTL, to verify that the response does not need to be immediate. Since an immediate response could be interesting to be verified, based on several works [42,51], we have specialized the *Response* pattern into another two subpatterns: *Eventual response* and *Immediate response*, as stated in Table 4. We note the difference between the *Temporal implication* pattern (*Liveness*) and the *Response* pattern, since while the *Temporal implication* pattern refers to that p is possibly followed by q, the *Response* pattern refers to that p is necessary followed by q [43]. As *example*, in the context of a lymphoma treatment guideline, authors in [27] use an *Eventual response* Property to verify that "there is a treatment in which growth factors are administered, when leukopenia appears', represented in LTL as G(leukopenia_value = present → F(done = growth_factors_administration)). Similarly, we could define a Property to check whether the administration of such growth factors is immediate (thus obtaining an *Immediate response* property). In [25], another two *Eventual response* properties are given within the medical domain.

Constrained chain. Finally, we have decided to consider a new *Order* pattern named *Constrained chain* which was included on Dywer et al.'s patterns [52] after we presented our work in [6]. This pattern basically presents a constrained chain of propositions, informally read as "q1, q2 without q3 responds to p", and whose definition is presented in Table 4 (adapted from [52,53]).

As a result, our final Property specification pattern hierarchy built upon our previous one can be seen in Fig. 7. In particular, we have both included new patterns (represented on a square), and specialized several patterns from our previous proposal (depicted on a double square). To sum up, we emphasize that since the proposed hierarchy of patterns gives support for representing the complete list of 45 requirements collected from the analyzed papers, we think that this approach is sufficiently complete for representing a wide spectrum of guideline properties.

4. Verification process based on constraint logic programming

As described previously, in [6] we chose SPIN to check clinical guidelines against properties specified in LTL. However, while most of the patterns included in our proposal can be specified in LTL (both the set of patterns given in [6] and its extension presented in the previous section), properties of *existential nature* can not be directly represented in the LTL language [6]. Among such properties, those conforming to *Possible existence*, *Chain Possible Existence with* and *without order* and *Possible Absence* can be verified with model checkers such as SPIN by means of the verification of the negation of the Property [6]. However,

as we stated in [6], commonly used properties such as the ones included in the *Liveness* pattern are not supported by LTL. This is the case of, for example, the *Reachability* pattern, whose associated formula in CTL is `AG(EFp)` and neither this formula nor its negation are represented in LTL. Other examples are properties conforming to the *Temporal implication*, *Everywhere eventually* and the *Always eventually* patterns. Aimed at providing support for the verification of properties which could not be verified with our previous approach based on SPIN, we have looked for other alternative formal methods.

Recently, we have performed an extensive research work on the verification of system models using model finders. More specifically, we have used a model finder to reason about UML modeling foundations. Such a work has been performed through two lines considering both static UML models (in particular, UML Class diagrams) [54–56], and dynamic UML models (UML State Machine diagrams) [57]. Taking into account that our proposal for the representation of guidelines is based on UML State Machines, together with our experience in using model finders to reason about UML models (both static and dynamic), we have decided to apply this paradigm to the verification of guidelines as a complementary technique to the one we proposed in [6], yielding positive results. In particular, we use the model finding and design space exploration tool *Formula*, which is based on Algebraic Data Types (ADT) and Constraint Logic Programming (CLP) [10,58,59]. As we will see later, and stated in [60], CLP provides us with an excellent framework for specifying and verifying properties of reactive systems (in our case, clinical guidelines). Next, we justify our choice of Formula and present our proposal to verify requirements in guidelines based on model finders.

4.1. Using a CLP-based model finder as a complementary technique for guidelines' verification

The different types of formal reasoning, such as *model checking* and *model finding*, can be understood, as stated in [61], by the *models relation* $\models_{\mathcal{T}h}$ (also known as "entailment"). The *models relation* $\mathcal{M} \models_{\mathcal{T}h} p$ pairs a structure $\mathcal{M}$ with a formula p whenever the structure gives a valid interpretation to the formula under the theory $\mathcal{T}h$. In the case of model checkers is to decide if $\mathcal{M} \models_{\mathcal{T}h} p$, given $\mathcal{M}$ and p. On the contrary, in the case of model finders is to generate a model $\mathcal{M}$ given a formula p such that $\mathcal{M} \models_{\mathcal{T}h} p$. More specifically, whereas model checkers take a structure $\mathcal{M}$ and a specification p, usually in the form of a temporal logic formula, and check whether $\mathcal{M}$ is a model of p, model finders take a logic formula p and attempt to find a model $\mathcal{M}$ of it [62]. Not only do model finders help find instances of the model specification and check it for contradictions, but they also enable to check user specified assertions against the specification [63].

Taking into account our previous proposal for verifying clinical guidelines using model checkers (SPIN), in this paper we complete it by giving the possibility of using a model finder. In particular, we advocate for using the Formula tool [10], as model finding and design space exploration tool, and the Formula language, for the semantics preserving translation of the models to be verified. Starting from a system specified in the Formula language (from now on, the *system's model design* or simply *design*), Formula allows us to explore the system design space. More specifically, a *system design space* refers to the number of potential design variants for the system. Thus, design space exploration refers to the act of considering possible options for any decision that contributes to the construction of a system. Based on our concrete context, and as we will explain later, Formula will allow us to reason about clinical guidelines by exploring different decisions regarding their design. As mentioned previously, Formula is based on ADT and CLP. CLP or *Constraint Logic programming* provides a powerful approach to writing formal specifications. CLP began as a merger of two declarative paradigms: constraint solving and logic programming. This combination helps make CLP programs more expressive and flexible, while in some cases being more efficient than other kinds of programs. More specifically, a *CLP model* is described via a set of variables, a set of clauses with constraints over them, and a goal to be satisfied. The clauses in the model are just restrictions imposed over the combination of values of some variables of the model. Solving a CLP model means finding a way to assign values to all its variables such that the goal given by all constraints is satisfied [9,58]. In particular, as stated in [61], in Formula, models $\mathcal{M}$ are finite sets of well-typed terms and the theory $\mathcal{T}h$ is a subclass of fixed point logic (FPL) [64]. More specifically, Formula is a constraint logic programming language based on fixed point logic over Algebraic Data Types. Based on an initial set of facts specified using Algebraic Data Types and a set of inference rules (Formula *rules* as we will explain later), Formula can deduce a set of final facts that is the least fixed point solution for the specifications.

Regarding Formula's characteristics, this tool has been proven to present distinctive strengths compared to other similar tools, including better expressiveness [58,65]. More specifically, Formula has two main characteristics that would help us to achieve our expectations:

- *World search*. Based on ADT and CLP, this tool relies on the *Formula solver Z3* as underlying engine to reason about models where proof goals are encoded as a CLP satisfiability problem. Formula utilizes a bounded verification approach by means of which the reasoning process is carried out by establishing finite bounds for the number of instances of the model to be considered during the verification process. In the case that *Z3* finds a solution that satisfies all encoded constraints (represented by Formula *queries*), Formula will reconstruct a complete model from this information derived from known facts (see next point *partial models*).
- *Partial models*. Sometimes the user could have partial knowledge about the model he/she wants to obtain. Such a knowledge can come from sources such as requirements, existing inputs, known faults, or concrete execution traces. Formula gives support for partial state completion, that is, using *partial model* knowledge to help generate the desired model. In this way, users can specify a small partial portion of the model directly representing their verification intent.

Taking into account these two Formula characteristics, using Formula for reasoning about clinical guidelines represented as state machines results in supporting several scenarios that enrich our previous proposal in [6]. Next we describe such scenarios, which have been adapted from [36] to be applied to the particular case of guidelines' application. In our explanations, we use the term *verification concern* to refer to either (1) a *requirement* or *correctness property* to be checked in a guideline (which can be specified by following our Property specification patterns and which will represented in our proposal by means of Formula *queries*) or (2) *a partial portion of the guideline's execution trace* the user wants to be considered in the reasoning process as part of the solution (that is, the desired overall valid execution trace). In this latter case, we would like to note that such a partial portion of the execution trace will be represented in our proposal by means of customized *partial Formula statements*. As we will see later, such *partial statements* will refer to already executed actions, that at the same time, represent concrete state configurations in the guideline's application. Taking this into account, the scenarios are the following:

- *Consistency checking- Does there exist a legal model?*: whether there exists a legal model of the guideline, that is, an execution trace that starts from the initial state, moves from state to state and reaches a final state (the guideline exhibits at least a valid execution trace). Such a legal model is necessary in order to build trust into the defined guideline.
- *Synthesis- Does there exist a model that has a certain property?*: if there exists a valid execution trace that satisfies a certain *verification concern*. Formula can be used to rigorously reason about the guideline's model design, by checking predefined correctness

properties about the execution traces (including, not only properties that could be verified with our previous proposal based on SPIN, but also not supported types of constraints) or ensuring that there exists a valid execution trace that includes a set of *partial statements*.

- *Design space exploration- Do there exist many models that have a property?*: whether there exist many valid execution traces that satisfy a certain *verification concern*. The Formula representation of the guideline can be used to inspect the clinical guideline model represented by the state machine, in order to search for conforming execution traces and to choose those which satisfy the desired *verification concern*. It is worth highlighting that our previous proposal based on the SPIN model checker did not provide us with this possibility.

We would like to note that valid execution traces obtained from both *synthesis* and *design space exploration* scenarios, would have to correspond to legal guidelines' models (*consistency checking*).

As we have mentioned previously, we have already used Formula to reason about UML diagrams [54–57]. More specifically, while in [54–56] we presented a framework to reason about UML Class Diagrams aiming at finding sets of classes' instances conforming to a class diagram, in [57] we completed such a work to give support also to UML State Machine diagrams, where we focused on reasoning and verifying state machine designs by generating possible sets of state configurations. As for the reasoning of UML State Machine diagrams, we note that in [57] we just focused on proving the *reachability of states* and *consistency requirements* in state machines without focusing on a specific context, covering just several aspects provided by the Formula *synthesis* scenario. More specifically, regarding *consistency requirements* in [57], we refer to *consistency* from a structural perspective, referring to properties that the model is expected to satisfy irrespective of its semantic content. Thus, a *consistency requirement* could be considered as a *requirement* or *correctness property* that just focuses on structural aspects of the model. In this paper, we extend and complete such a proposal by considering the other two scenarios, basing also on concrete semantics provided by state machines representing guidelines.

4.2. Proposed verification process

The reasoning process of a clinical guideline using Formula follows the steps described on top of Fig. 1. Next, we describe these steps in detail.

Step 1. Encoding the Clinical Guideline into Formula. Starting from a clinical guideline represented as a state machine as specified by the guideline to State Machine representation patterns described in [6] (identified as *Step 0*), we first translate such a state machine model into a Formula specification model by performing an intermediate step (identified as *Steps 1a* and *1b*). Our proposal for representing clinical guidelines as UML state machines [6,13] considers the possibility of using complex structures like *simple composite* states, *orthogonal composite* states, and *submachine* states to represent guidelines' states. Later, the verification of the guideline can be performed directly on the resulting state machine containing such complex structures, but it requires rather complex strategies, because such structures complicate the traversal and analysis of the state machine. A commonly used alternate approach to tackle state machines is to flatten the state machine first, by removing hierarchy and concurrency (*simple composite*, *orthogonal composite* and *submachine* states) so that all states are atomic [66], and then apply the required strategy (such as analysis, verification or code generation processes) [67].

Although there are many techniques and algorithms reported in the literature to flatten state machines [68], an extended strategy to eliminate *hierarchy* consists on handling transitions that leave and enter composite states. In particular, transitions that leave a composite state are transferred to its subtates, while transitions that enter a composite state are redirected to the initial substate of the composite state. On the other hand, the elimination of *concurrency* would be done with the cartesian product of all states (and transitions) in each concurrent region. We note that, if the original state machine model has concurrent states, it is said that such an operation causes an explosion of the number of states and transitions in the resulting model [69]. For this reason, the complexity of flattened models has been studied previously, especially within the model's verification community, where different questions relevant to the verification process have been addressed. However, conclusions drawn from such studies are many and varied because of the different nature of the verification tool used in each study (and thus, the tool's language semantics). For example, while some works stated that "it may be beneficial to maintain the state machine hierarchy in the encoding instead of flattening the state machine before encoding" [70], others stated that flattening provides a better separation of concerns and a lesser analysis complexity [67]. In this context, several proposals have been given to avoid states and transitions explosion [71,72].

In our particular case, we have considered the possibility of either providing a proposal that handled hierarchies and concurrency without flattening them, or adopting a preprocessing step which flattened the source guideline's state machine. Based on our previous experience with Formula [54–57], representing hierarchy and concurrency would lead to complex Formula constructs, thus entailing an unnecessary overload of the Formula solver. We also note that previous work on representing guidelines published by the Formula's authors [36] only provided support for simple states. Taking this into account, we have decided to flatten the state machine representing a guideline so that we get a better separation of concerns, obtaining a model which can be more easily tackled by Formula. Thus, our translation proposal from a clinical guideline represented as a state machine to Formula constitutes a stepwise process so that the state machine is first flattened by applying a flattening algorithm (see *Step 1a*) and, second, the resulting state machine is translated into Formula by following a set of translation patterns we have defined (see *SM2F translation patterns* in *Step 1b*). In particular, regarding *Step 1a*, we can use one of the many techniques and algorithms reported in the literature to flatten concurrent and hierarchical states (see [68] for a complete mapping study of flattening techniques). *Step 1b* will be described in detail in Section 6.

Step 2. Specification of Verification Concerns. Second, each *verification concern* (*requirement* or *partial model statements*) desired to be considered for the verification of the guideline would have to be manually defined in the Formula language and included in the guideline's Formula specification (see *Step 2*). This step will be explained in detail in Section 7.

Step 3. Verification of the Clinical guideline in Formula. Finally, the Formula tool takes the model resulting from the previous step to reason about the guideline (see *Step 3*). The results of the verification process can be many and varied. Different examples will be presented in Sections 7 and 8, where the specification of concrete *concerns* to be verified in the *AP guideline* and the verification process itself are described, respectively. It is worth noting that in case the feedback given by the Formula tool refers to inconsistencies detected in the guideline's definition, such information could be considered, for example, for the redefinition of the guideline as appropriate.

We would like to note that the two verification processes depicted in Fig. 1 aim to constitute an overall framework for the verification of guidelines sharing the first common step of representing guidelines by means of state machines. Before going on to describe in detail the verification process presented in this paper, next we give a brief overview of Formula syntax and semantics. For a more detailed description, we refer the reader to the Formula Website [10] or to specific papers [59,64].

5. A brief overview of Formula

5.1. Formula main units

Formula allows representing a system's design by using three

```
1 domain MetaLevel {
2    *********Enumeration types*********
3    stateType ::={final, simple}.
4    pseudostateType ::={initial, choice}.
5    transitionKind ::={internal, local, external}.
6  **********Data types**********
7    [Unique(name ->type)]
8    primitive State ::= (name: String, type: stateType).
9    [Unique(name -> type)]
10   primitive Pseudostate ::= (name: String, type: pseudostateType).
11   Vertex ::= State + Pseudostate + ....
12   [Closed(src, dst)]
13   primitive Transition ::= (name: String, type: transitionKind,
                               src: Vertex,
          ...                  dst: Vertex).
14   conforms := ... . //defined queries
15 }
```

Fig. 8. An extract of a Formula domain.

different units: *domains*, *models* and *partial models*. Firstly, a *Formula domain FD* is the basic specification unit in Formula and is used to formalize an abstraction of the problem that can be used by Formula to reason about the design. This type of units allows specifying algebraic data types and a logic program describing properties of the abstraction. The CLP paradigm provides a formal and declarative approach for specifying such abstractions [10], which in Formula are represented by *rules* and *queries* (which we will explain later). As an example, Fig. 8 shows the definition of a domain called `MetaLevel` containing an algebraic data type named `State` in line 8. Additionally, domains can extend other domains by including the `extends` keyword. A *Formula model FM* is a finite set of data type instances built from constructors of the associated domain *FD* which satisfies all its constraints [59]. As an example, the Formula expression `State("ReadyToBeAliquotedS1", simple)` would correspond to an instance of the data type `State` described previously. Formula allows specifying individual concrete instances of the design space or parts thereof, in a specific Formula unit called *partial model*. A Formula *partial model* FPM is a set of instance-specific facts placed along with some explicitly mentioned unknowns (also known as *fresh variables*, denoted as '_'), which correspond to the parts of the FM that must be solved [10]. Partial models allow unknowns to be combined with parts of the model that are already fixed [10,59]. They are essentially lower bounds on the type of models we want to find. Fixed parts of a model can be included in the partial model explicitly, specifying the corresponding Formula statements inside the model. Additionally, it is necessary to specify the domain(s) the partial model conforms by using the `of` keyword.

As described previously, a Formula *domain FD* consists of *algebraic data types*, *rules* and *queries*. First, *algebraic data types* constitute the key syntactic elements of Formula. Based on the defined data types, *rules* and *queries* are specified as logic program expressions ensuring the remaining constraints [10]. In general, *rules* specify implications and *queries* allow representing both forbidden and valid states. Next, we explain the main characteristics of these constructors.

5.2. Formula data types, rules and queries

Algebraic data types. They are defined by the operator :=, indicating on the right hand side their properties by *fields*, which must be of some concrete type (Formula built-in types or other user data types). A data type definition can be labeled with the `primitive` keyword, denoting that it can be used for the extension of other type definitions (thus, defining *primitive constructors*). In addition to the `State` data type, previously cited, in line 13 of Fig. 8 we define the `Transition` data type, which represents the `Transition` element of the UML State Machine metamodel. This type defines several *fields* together with their types (such as the fields `src` and `dst` of type `Vertex`, representing the source and target vertexes of a transition, respectively). If the `primitive` keyword is omitted, the data type definition results in a *derived constructor*. As an example, see the definition of the type `Vertex` in line 11, representing the `Vertex` element of the UML metamodel (which includes elements such as *states* and *pseudostates*). Additionally, constants are defined using the operator :=, specifying a fixed value or a list of fixed values within curly brackets. For example, the constant `stateType` in line 3, would represent different types of states (i.e., an enumeration of fixed values.

Around data types, Formula defines different categorizations of the structural elements as building blocks for defining Formula expressions. These structural elements are mainly Formula *terms*, among which we note *simple* and *compound*. A *simple term* is represented by means of a type identifier containing variables, constants, or other simple terms as arguments, within parenthesis. A *compound term*, on the other hand, is represented by means of a type identifier with a list of *terms* within parenthesis. An example of a *simple term* is `State(x,y)`, representing a specific state with name 'x' and type 'y' (see the definition of the `State` data type in line 8). With the *compound term* `Transition(_,_,State(x,y),_)`, we represent all the instances of the `Transition` term where the third field is set to a fixed Property (`State(x,y)`). The other fields are filled with an unknown do not-care symbol ('_'), so that Formula can find valid assignments. In this way, this term represents any transition whose source state is the specific state (`State(x,y)`).

In particular, Formula *terms* are, directly or indirectly, the basis for constructing *predicates*, which constitute basic units of data, used for defining *queries* and *rules*. For example, predicates can be constructed from *compound terms*, described previously, and *bindings*, that is, gluing a variable to either a type expression or a *compound term*. An example of a binding term is `t is Transition(t1,_,State(x,y),State(z,w))`, where the variable `t` is bound to the compound term `Transition(t1,_,State(x,y),State(z,w))`, so that the identifier `t` stands for such a term.

Additionally, Formula allows using different *annotations* in the definition of data types to *reduce the size of the search space*. For example, the `[Closed]` annotation, whose syntax is `[Closed(DT fields)]`, which instructs Formula to apply a closed check to instances of the corresponding data type (`DT`) fields that are using only existing instances of types associated with these fields. Otherwise Formula would be able to invent new instances, which is a desired behavior for general model-finding problems. An example of the `[Closed]` annotation is illustrated on line 12 in Fig. 8 where it ensures that Formula instances of transitions are created by state instances that exist in the model. Additionally, `[Unique]`, whose syntax is `[Unique(DT fields -> DT fields)]`, requires all records with identical fields on the left of the arrow (->) to have identical fields to the right of the arrow. By adding the `[Unique]` attribute to a type constructor definition, Formula introduces new queries to the containing domain, which ensure that an element of the domain of the relation is mapped to a single element of the codomain. As an example, the `[Unique]` annotation on line 7 in Fig. 8 checks that there are not two state instances with the same field values.

Rules. A *rule* behaves like a *universally quantified implication*, that is, whenever the relations on the right hand side of a rule hold for some substitution of the variables, then the left hand side holds for that same substitution [59], and Formula will generate a new entry in the factbase of Formula corresponding to the left hand side (thus, rule's main aim is *production*). Rules are specified by the operator :-, indicating, on the left hand of the expression, a *simple term* and, on the right hand, the list of *predicates* specifying the rule (an example of a rule is shown in next section).

Queries. A *query* correspond to a rule where left hand side are nullary constructors [58] (that is, constructors that take no arguments). In contrast to a Formula rule, a *query* does not add new terms into the factbase of Formula, but behaves like a proposition variable that is true if and only if the right hand side of the definition is true for some substitution [59]. Queries are constructed using the operator :=, joining Formula *predicates*. In particular, Formula defines in every domain the `conforms` standard query that combines other queries using logical

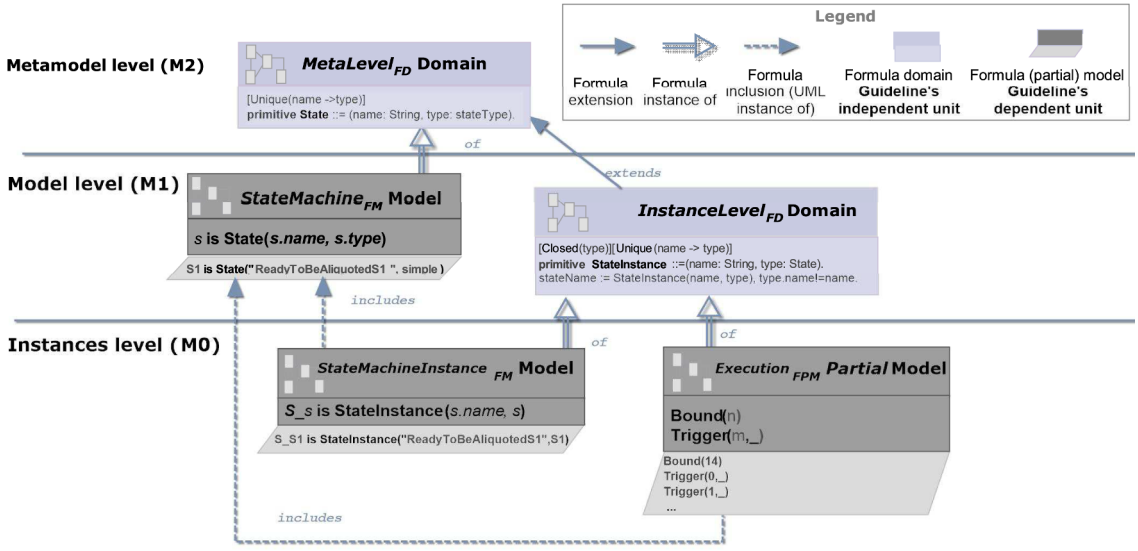

Fig. 9. An excerpt of the Formula units created for the representation of our case study in Formula.

operators and is used as the main goal to find a model which conforms the domain. When a (partial) model is inspected in Formula, the `conforms` clause is the starting point of the searching procedure. If it is not possible to find an instance that satisfies this special query, the (partial) model is said to be *Unsatisfiable*. Based on the *existential quantification* semantics of queries, we can use them to prove the existence of specific facts in the model. Additionally, the *universal quantification* can be achieved by verifying the negation of a query representing the opposite of the original predicate. For example, let's suppose that we want to ensure, by using Formula queries, that there exists a model in which `Transitions` are not created as connections of undeclared `States`. Then, we first need to define a query `q` representing the existence of transitions verifying the opposite (that is, `q := Transition(_,_,State(x,y),_), fail State(x,y)`). More specifically, we consider a `Transition` term defined with an undeclared source state (see `fail State(x,y)`, where the Formula keyword `fail` means negation as failure). With this query we are considering such incoherent situation as a valid state. Thus, to verify that such situation is not valid, we need to include the negation ('!') of the query (`!q`) in the `conforms` query of the specific domain. If the verification process results in a model (*existence quantification*), all its `Transition` terms would be defined with declared `State` terms. If we were interested in proving the *universal quantification*, that is, in all models `Transition` terms are created as connections of declared `State` terms, we would need to verify directly `q`, so that an *unsatisfiable* result would prove that there not exists any model where `Transition` terms are created as connections of undeclared `State` terms. We would like to note that queries can be used to represent both forbidden and valid states; it depends on the context, since in some occasions a query could represent a forbidden state (thus, failing to verify such a query would be desired), while in other occasions a query could represent a valid state (and its successful verification would be desired). A more complete discussion of these aspects is given in Section 7, where we describe the specification of *requirements*.

Finally, having defined the domains, models and partial models for the specific problem, Formula explores the design space by loading such units, and executes the logic program. It finds all intermediate facts that can be derived from the given facts in the partial models, and tries to find valid assignments for the unknowns [10,59].

6. Encoding clinical guidelines in Formula

In this section we first explain our proposal for the representation in Formula of a UML state machine representing a clinical guideline. At this point, we would like to note that, while in [57] we considered state machines representing the dynamics of simple systems without focusing on a specific context, in this paper we extend our previous proposal by giving support to more complex state machines' semantics implicit in clinical guidelines represented as state machines. We finish this section by describing our proposal for the automatic translation of a guideline's state machine to Formula, by means of which all the units that constitute the Formula specification of the guideline are automatically generated from the guideline's state machine.

6.1. Overview of our translation proposal

Our proposal for encoding a clinical guideline into Formula follows a MOF-like metamodeling approach [57]. More specifically, we propose the Formula specification of a guideline to be constituted of five different Formula units distributed along the MOF Metamodel (M2), Model (M1) and Instance (M0) levels [7], described in Section 2: the *MetaLevel* and *InstanceLevel* domains, the *StateMachine* and the *StateMachineInstance* models, and the *Execution* partial model. In order to have a better understanding of our proposal, in Fig. 9 we illustrate the defined units, together with the relations among them. To help the reader better understand our approach we focus explanation on the specific translation of *states* and, in particular, the state *ReadyToBeAliquotedS1* of our case study. More specifically, in this figure the five Formula units are represented by rectangles, which include the transformation patterns defined in each case for representing states, while the Formula expressions resulting from the application of such patterns to our concrete guideline are depicted by rhomboids.

Taking into account the semantics of *Formula domains*, *models* and *partial models*, we have decided to define the following five units. First, we represent the UML State Machine's constructs by means of two domains in order to provide an abstraction of elements at the *meta* and the *instance* levels (see *MetaLevel* and *InstanceLevel* domains in Fig. 9, respectively). Since these domains are the same whatever guideline's state machine is being translated, they are *guideline-independent*. Regarding models, we define the *StateMachine* and the *StateMachineInstance* models, basically aimed at representing the concrete conceptual and instance elements of a specific state machine, respectively. That is, in contrast to the defined domains, these two models are *guideline-dependent*, as they represent concrete aspects of the guideline at hand (see Fig. 9). So that Formula can take the previous elements and organize them into valid execution state configurations, we have defined a

Table 5
Excerpt of the proposal regarding the creation of the Formula domains.

Domain	Level	State	Transition	Pseudostate	Vertex
$MetaLevel_{FD}$	*M2*	[Unique(name ->type)] primitive State :: = (name: String, type: stateType).	[Closed(src, dst)] primitive Transition :: = (name: String, type: transitionKind, src: Vertex, trg: Vertex).	[Unique(name -> type)] primitive Pseudostate :: = (name: String, type: pseudostateType).	Vertex :: = State + Pseudostate +....
$InstanceLevel_{FD}$	*M1*	[Closed(type)][Unique(name -> type)] primitive StateInstance :: = (name: String, type: State). stateName: = StateInstance(name, type), type.name! = name.	[Closed(type,source,target)] [Unique(name-> type)] primitive TransitionInstance :: = (name: String, type: Transition, source: VertexInstance, target: VertexInstance). transitionName: = TransitionInstance(name,type,...), type.name! = name.	[Closed(type)][Unique(name -> type)] primitive PseudostateInstance :: = (name: String, type: Pseudostate). pseudostateName: = PseudostateInstance(name,type), type.name! = name.	VertexInstance :: = StateInstance + PseudostateInstance...

partial model *FPM* called *Execution* (see Fig. 9).

Next, we explain how to create such Formula units, relying on Table 5 (to explain the Formula statements that constitute the domains) and on Table 6 (to present the translation algorithms defined to generate the Formula models). In this latter table, we represent in bold font the fixed elements in the translation, that is, parts of the Formula instructions that keep the same independently of the concrete state, transition or pseudostate being translated, respectively (that is, parts shared by the translation of all states, transitions or pseudostates, respectively). Statements in Table 5 are fixed. We do not provide a table for the *Execution* partial model because, as explained later, its definition is straightforward.

6.2. Formula data types

The defined domains *FD MetaLevel* and *InstanceLevel* (see Table 5) contain (1) specific data types, which allow us to represent facts and generate reasoning instances of such types, and (2) queries, which restrict the valid system states by specifying forbidden states.

MetaLevel domain. For each metamodel element *State*, *Transition*, *Pseudostate* and *Vertex*, we mainly define a primitive Formula data type with the same name and specific fields. An explanation of most of the defined elements has been provided in the previous section since Fig. 8 presents an excerpt of this domain. The definition of these types will allow us to create Formula instances representing specific UML *States*, *Transitions* and *Pseudostates* at the Model level (such as the specific state *ReadyToBeAliquotedS1*, identified as `S1`, shown in the first row of Table 6).

InstanceLevel domain. To be able to represent the information generated during the execution of a state machine (that is, the state configurations which constitute the execution traces, together with the representation of the triggered transitions), we have defined specific types included in a Formula domain *InstanceLevel*. In the second row of Table 5, we show the concrete data types to be defined and which are associated to *state*, *transition*, *pseudostate* and *vertex* elements (see the different columns). For example, this domain defines types such as `StateInstance` or `TransitionInstance`. We would like to note that we have defined three different Formula queries (see `stateName`, `transitionName` and `pseudostateName` in Table 5) to ensure that the name of a state/transition/pseudostate instance is the same as the name of the state/transition/pseudostate to which such an instance corresponds, respectively (note that it refers to *universal quantification*). For example, in the case of states, firstly we need to define a query representing the existence of a `StateInstance` element verifying the opposite, that is, its name does not match the name of the `State` type to which it corresponds (see query `stateName` in Table 5). With this query, we are considering such incoherent situation as a valid state. Thus, to verify that such a situation is invalid, we finally include the negation ('!') of the query in the `conforms` query (not included in Table 5).

6.3. Formula data types' instances

Once we have defined the Formula domains with the abstractions of the problem, as mentioned previously, Formula gives us the possibility of creating a model FM as a finite set of data type instances built from constructors defined in the domains *FD* which satisfies all the *FD* constraints [59]. In our particular case, we have defined two different Formula models which, as advanced previously, are *guideline's dependent*. For this reason, in Table 6 we present our proposal for the definition of these models by using translation algorithms.

StateMachine model. This model, which contains the instances of the data types created in the *MetaLevel* domain, represents the specific elements of a particular state machine (see Fig. 9). More specifically, for each specific *state*, *transition* or *pseudostate* element in the guideline's state machine, we define a Formula *predicate* representing an instance

Table 6
Excerpt of the proposal regarding the creation of the Formula models.

Model	Level	State	Transition	Pseudostate
$StateMachine_{FM}$	*M1*	*Translation algorithm:* *for_each s state in Guideline's state machine* if s.isSimple then stateType = "simple" else_if s.isComposite then stateType = "composite" else s.isOrthogonal stateType = "orthogonal" end_if s **is State**(s.name, stateType) *end_for_each* *Example:* **S1 is State**("ReadyToBeAliquotedS1", simple) **S2 is State**("InFillingProcessS2", simple) **S5 is State**("AliquotedS5", simple) **FINAL is State**("final", simple)	*Translation algorithm:* *for_each t transition in Guideline's state machine* t **is Transition**(t.name, t.type, t.source, t.target) *end_for_each* *Example:* T0 **is Transition**("t0",external,INI,S1) T1 **is Transition**("t1",external,S1,S2)	*Translation algorithm:* *for_each p pseudostate in Guideline's state machine* p **is Pseudostate**(p.name, p.type) *end_for_each* *Example:* INI **is Pseudostate**("ini",initial)

$StateMachineInstance_{FM}$	*M0*	*Translation algorithm:*
		for_each s state in Guideline's state machine
		S_s **is StateInstance**(s.name, s)
		end_for_each
		
		Example:
		S_S1 **is StateInstance**("ReadyToBeAliquotedS1",S1)
		S_S2 **is StateInstance**("InFillingProcessS2",S2)
		S_S5 **is StateInstance**("AliquotedS5",S5)
		S_FINAL **is StateInstance**("final",FINAL)
			Translation algorithm:
			for_each t transition in Guideline's state machine
			if t.source.isKindOf(UML.State) then
			sourceType = 'S_'
			else
			sourceType = 'P_'
			end_if
			T_t **is TransitionInstance**(t.name, t, sourceType + t.source, targetType + t.target)
			end_for_each
			
			Example:
			T_T0 **is TransitionInstance**("t0",T0,P_INI, S_S1)
			T_T1 **is TransitionInstance**("t1",T1, S_S1,S_S2)
				Translation algorithm:
				for_each p pseudostate in Guideline's state machine
				P_p **is PseudostateInstance**(p.name, p)
				end_for_each
				
				Example:
				P_INI **is PseudostateInstance**("ini",INI)

of the corresponding constructor (`State`, `Transition`, or `Pseudostate` in the *MetaLevel* domain), where the name of each predicate's variable corresponds to the element's identifier (see first row in Table 6). With these Formula instances, we are explicitly representing specific elements in the guideline's state machine. For example, in the first column of Table 6 we show the definition of the predicate `S1 is State("ReadyToBeAliquotedS1", simple)` representing the state *ReadyToBeAliquotedS1*, which corresponds to a Formula instance of the constructor `State` defined in the *MetaLevel* domain.

StateMachineInstances model. This model contains the instances of the data types defined in the *InstanceLevel* domain. In particular, such instances refer to the *state*, *transition* and *pseudostate* instances that Formula would use as constructors of the state configurations (execution traces) of the specific clinical guideline. More specifically, for each specific *state*, *transition* or *pseudostate* in the guideline's state machine, we define a Formula *predicate* representing an instance of the corresponding data type defined in the *InstanceLevel* domain (see second row in Table 6). The name of each predicate's variable corresponds to the prefix `S_`, `P_` or `T_` (depending on whether it refers to an state, transition of pseudostate element, respectively) plus the element's identifier `id` (see Table 6). For example, in this model we define predicates such as `S_S1 is StateInstance("ReadyToBeAliquotedS1",S1)`, which would represent the fact that a specific aliquot object has been in the state *ReadyToBeAliquotedS1* (see first column in Table 6). On the top of Fig. 10 we show graphically the overall instances we would define for the case study.

We would like to note that an instance of some type in the state machine is translated into the identifier of this instance and vice versa and the same identifiers of states, transitions and pseudostates are used in UML state machines and Formula units (for example, the identifier of state *ReadyToBeAliquotedS1* in the state machine, that is, `S1`, is used as identifier of its translation, that is, `S1 is State ("ReadyToBeAliquotedS1", simple)`.

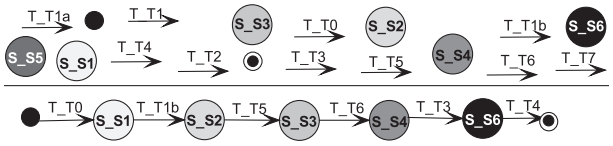

Fig. 10. Instance elements and complete execution trace.

Taking this into account, the *StateMachine* model conforms to the *MetaLevel* domain, while the *StateMachineInstances* model conforms to the *InstanceLevel* domain.

6.4. Logic statements to simulate a guideline's application

Up to now, we have established the bases to represent in Formula (1) the UML State Machine metamodel, (2) specific state machines (that is, guidelines) conforming to such a metamodel, as well as (3) the concrete instances produced during the execution of a state machine (that is, the guideline's application) which would constitute each state machine execution trace (that is, the sequence of state configurations an object can go through during the guideline's application). Nevertheless, the defined Formula data types, instances and queries are not enough to allow Formula to reason about the state machine execution, that is, to take such concrete elements and organize them into valid execution state configurations. More specifically, in addition to such instances (see the top of Fig. 10) and queries, we provide Formula with specific data types and rules to instruct the tool in the way to reason about such data, so that it is able to generate valid execution traces (such as the one shown on the bottom of Fig. 10, which corresponds to trace labeled (b) in Fig. 2). For this reason, we have completed our proposal by defining new Formula statements in the *InstanceLevel* domain, and introducing the *Execution* partial model (we note that the new statements in the *InstanceLevel* domain include several improvements related to our

```
1 domain InstanceLevel extends MetaLevel {
2  ....
3  primitive Trigger  ::=(t: Natural, tr: TransitionInstance ).
4  primitive Bound::=(n: Natural).
5  StateConfiguration  ::= (t: Natural, v: VertexInstance , traT: String).
6  ChoiceGuardValue  ::= (guardValue: String, trName: String, ch:VertexInstance ).
7  StateConfiguration (0,ini, traT):- ini is PseudostateInstance ("ini",INI), traT="__".
8  StateConfiguration  (tnext, tgt,tn) :-
                        ini is PseudostateInstance (_,Pseudostate(_,initial)),
                        StateConfiguration (t, ini,tna),
                       Trigger ( t,TransitionInstance (tn,_,ini,tgt)),
                        tnext=t+1, Bound(end),  t<end.
9  StateConfiguration  (tnext, tgt,tn) :-
                        src is  StateInstance (_,_) ,
                         StateConfiguration (t, src,traT),
                        Trigger ( t,TransitionInstance (tn,_,src,tgt)),
                         tnext=t+1, Bound(end),  t<end.
10 ChoiceGuardValue  (tgt.name,tn,choice),
   StateConfiguration  (tnext, tgt,tn) :-
                       choice is  PseudostateInstance (_,Pseudostate(_,choice)),
                        StateConfiguration (t, choice,tna),
                       Trigger ( t,TransitionInstance (tn,_,choice,tgt)),
                        tnext=t+1, Bound(end),  t<end.
11 done:=  count( StateConfiguration (_,StateInstance("final",FINAL),_))=1.
12 conforms:=   .... & done.
13}
```

Fig. 11. Generation of new state configurations.

previous proposal in [57], as we will remark in this section).

New statements in the InstanceLevel domain. We need to indicate Formula the way in which it has to generate a valid chain of state configurations which will constitute the valid execution traces. As described in Section 4.1, Formula utilizes a bounded verification approach by means of which the reasoning process is carried out by establishing finite bounds for the number of instances in the model to be considered during the verification process. Thus, we first need to provide Formula with an upper bound representing a limit in the number of triggered transitions conforming to such valid execution traces. For this reason, first, we have included two new data types: `Trigger` and `Bound`. On one hand, the `Trigger` type has been defined to simulate the triggering of a transition (see line 3 in Fig. 11). For this reason, its definition includes a field `t`, referring to the time in which the triggering of the transition takes place, and the associated `TransitionInstance` instance, referring to the transition triggered (for example, `Trigger (3,T_T5)` represents that transition `T_T5` has been fired at time $t = 3$). On the other hand, the `Bound` type (see line 4 in Fig. 11) defines a *natural* number field `n` which would represent an upper bound to limit the number of transitions triggered during the state machine execution. Based on the `Trigger` type, we define the type `StateConfiguration` to represent each active state configuration (see line 5), and which has three fields: (1) `t`, which keeps track in time of the sequence of state configurations, (2) `v`, which refers to the specific active vertex, and (3) `traT`, which refers to the name given to the `TransitionInstance` instance corresponding to the transition which has been triggered to change to that state (for example, `T_T1`, in the binding term `T_T1 is TransitionInstance(t1,T1,S_S1,S_S2)`).

Additionally, in order to construct the chain of state configurations as the transitions are triggered, we have defined four different Formula rules (see lines from 7 to 10 in Fig. 11), in order to create new entries of type `StateConfiguration` in the fact-base of Formula. More specifically, we make a distinction depending on whether the state configuration to be created corresponds to: (1) the initial state configuration fact (line 7), (2) a state reachable from the initial state (line 8), (3) a state reachable from previous states (line 9), or (4) a state reachable from a choice pseudostate (line 10). For the definition of these rules we have taken into account the *production* semantics of rules. For example, with line 9 we state that, given the current state configuration `StateConfiguration(t,src,traT)` and the triggering of a transition `Trigger(t, TransitionInstance(tn,_,src,tgt)` (that is, the transition's source vertex corresponds to the current one), Formula nondeterministically creates a new fact `StateConfiguration (tnext,tgt,tn)`, which corresponds to the new state configuration where the new state `tgt` is the target vertex of the triggered transition. The value of the time parameter `tnext` is also incremented by 1 for the following state configuration. We also include the expression `Bound (end), t < end` (see line 9) to make sure that the number of transitions triggered during the state machine execution does not exceed the desired upper bound limit. Such an upper bound value will be established later, when defining the *Execution* partial model. We note that rules similar to these given in lines from 7 to 9 were already provided in [57], but in this paper we have enriched our previous proposal by redefining them so that Formula can more efficiently generate the `StateConfiguration` facts. Additionally, we have defined the rule in line 10 so that we also give support for generating `StateConfiguration` entries referring to states reachable from choice pseudostates, registering also the guard condition value that has triggered the transition to the new state configuration (see also the definition of the new type `ChoiceGuardValue` in line 6).

As another extension of our previous proposal, we include a new query called `done` which ensures us that Formula generates valid execution traces where the guideline finishes its application (see line 11). More specifically, in any case, the resulting chain of active state configurations corresponds to a guideline's execution which reaches a final state at some point, this ensuring well-formed execution traces. This query will be checked in the corresponding `conforms` query (see line 12).

Execution partial model. Following our strategy, each different state an object can go through during its lifetime is represented by a `StateConfiguration` fact, which is created by means of the rules defined from lines 7 to 10 in Fig. 11. Based on Formula rules' semantics, given a current object's state configuration (right part of any of those rules) we need suitable both `Trigger` and `Bound` facts, so that Formula can nondeterministically generate a new object's state configuration (left part of any of those rules). More specifically, first, we need to instruct Formula to find valid assignments for the `TransitionInstance` occurrences in the `Trigger` elements of these rules. As described previously, for this task Formula defines *partial models FPM* where we can specify individual concrete instances of the design space or unknown parts thereof (these latter corresponding to the parts of the model *FM* that must be solved by the Formula tool) [10]. Second, a `Bound(end)` term is defined, `end` being the desired upper bound value, representing the maximum length of an execution trace (for example, `Bound(12)` means that it can be at most 12 `Trigger` elements). Finally, we include such a `Bound(end)` term and as many `Trigger` terms as necessary in a partial model *FPM* called *Execution* (see Fig. 9). Each `Trigger` term defines a do not-care symbol ('_') in the field which corresponds to the `TransitionInstance` instance, so that Formula will find valid transition assignments.

At this point we would like to make a remark regarding the verification of user-defined *verification concerns* in the model (both *requirements* and *partial model statements*). More specifically, each *verification concern* has to be introduced to the appropriate Formula unit: while *requirements*, represented by means of Formula `queries`, are introduced to the *InstanceLevel* domain, *partial Formula statements* must be included in the *Execution* partial model (in the next two sections we will show examples of the verification of concrete *concerns*).

As mentioned previously, starting from the different Formula units defined for a specific guideline's state machine, including the desired user-defined *verification concerns*, Formula would explore the design space by loading such units, and execute the logic program. Formula finds all intermediate facts that can be derived from the given facts in the partial models, and tries to find valid assignments for the unknowns. The resulting solution, if exists, would refer to a well-formed execution trace satisfying all encoded constraints.

6.5. Automatic translation from a guideline to Formula

Aiming at automatizing the overall verification process as far as possible, we propose to use a tool chain which assists the user in

performing the encoding of a guideline into Formula. More specifically, since our clinical guideline to State Machine representation patterns [13] consider concurrency/hierarchy elements for guidelines' representation, first, a flattening tool can be used to obtain the guideline specification as atomic states (such as [73] or [74]).

As for the second step, we have based it on an Eclipse plug-in we developed in [57] to automatically transform general-purpose state machines to Formula specification based on the rules described in [57]. More specifically, this plug-in followed a MDD-based approach to implement such a transformation. The working philosophy behind MDD-based approaches focuses on models (in our case, a guideline's state machine model), so that the program code is automatically generated from them by means of a refinement process [75]. In particular, we have used the MOFScript Eclipse plug-in [76], which is a MDD-based tool with support for customizable model-to-text transformations. This plug-in implements the MOFScript language which was one of the candidates in the OMG Request For Proposal (RFP) process for MOF Model-to-Text Transformation [77]. Such a RFP identifies a set of high-level requirements that provide a framework for defining a language that fit the OMG way of thinking and align well with already adopted OMG specifications. Among such requirements, we note the ability of generating text from models (M1), specifying transformations based on metamodels (M2). We have implemented our proposed translation of state machines to Formula into the MOFScript language, resulting in an only set of MOFScript transformation files. Such transformations are based on the MOF M2 metamodel of UML State Machines, so that we are able to generate text (Formula specifications) from MOF M1-based models (state machines representing guidelines). The Eclipse plug-in that we defined in [57] includes such a set of transformation files and uses the MOFScript plug-in to execute them, so that a state machine can be automatically translated into the corresponding Formula specification. More specifically, firstly, the state machine model is created using any UML 2.0 compliant tool able to create models in the XMI format supported by EMF (such as UML2 Eclipse plug-in [78], or a graphical tool such as Borland Together Modeling tool [79]). The transformations defined in MOFScript traverse this input model and automatically generate the Formula specification.

As for the MOFScript transformations, in [57] we defined a main transformation (`main.m2t`) devoted to generate the final Formula specification file (`FormulaSpecificationsSM.4ml`) by invoking specific rules from the rest of the transformation files. This main transformation had a main rule which executed another three rules. First, the `createNonContextDependentUnits` rule created the *MetaLevel* and the *InstanceLevel* domain units, that is, the Formula units common to the translation of all guidelines. Second, the `getStatesTransitionsPseudostates` rule constituted the cornerstone rule of the transformation since it traversed the overall state machine looking for its content (*states*, *transitions* and *pseudostates*). Finally, the `createContextDependentUnits` rule used the information gathered by the previous rule to create the guideline-dependent Formula units (that is, the *StateMachine*, the *StateMachineInstance* and the *Execution* (partial) models).

Starting from the plug-in presented in [57], we have developed a second version to adapt it to the improvement suggestions made to the state machine to Formula transformation patterns described previously. Briefly speaking, we have mainly modified the definition of the rule `createNonContextDependentUnits` to include all the suggested changes, so that it generates: (1) the Formula rules described in lines from 7 to 9 in Fig. 11 as established in the new proposal, (2) the new Formula rule of line 10 in Fig. 11 that allows Formula to generate `StateConfiguration` entries referring to states reachable from choice pseudostates, (3) the new Formula type `ChoiceGuardVariable`, which allows us to represent guard condition values, and (4) the new Formula `done` query.

Finally, the defined MOFScript transformations have been integrated into another Eclipse plug-in so that the automatic transformation from a guideline's state machine to Formula can be performed from a menu item the plug-in provides. We have also improved this plug-in so that, by choosing such a menu, a GUI interface is dynamically created asking the user for the required value `n` of the `Bound` term in the *Execution* partial model. In particular, the inserted value is used to automatically generate such partial model with the `Bound` term, and as many `Trigger` terms as stated by such a value. Thus, once translation has been completed, the plug-in creates a file with the `.4ml` extension which includes all the Formula units constituting the Formula specification of the guideline.

7. Specification of requirements

Considering our two proposals for the verification of clinical guidelines based on model checking (SPIN) and model finding (Formula), respectively, and taking into account our extended hierarchy of Property specification patterns, we have analyzed the support each proposal provides for properties verification. More specifically, we have analyzed our extended hierarchy of patterns, looking for those properties which can be verified using one or another proposal. As a result, we can state that our previous SPIN-based approach is able to check properties conforming to most of the patterns (considering also the ones included in our extended patterns hierarchy), with the exception of properties conforming to the patterns: *Everywhere eventually, Always eventually, Liveness reachability* and *Temporal implication* (which can only be specified in CTL). Here is where our Formula proposal takes action, providing us with a way to verify such a type of properties. From now on, we will refer to properties conforming to such patterns as *selected existential* properties.

As for the verification of properties using Formula, we would like to note that, since this language does not provide us with temporal logic constructors (in particular, CTL constructors), the verification of temporal properties in Formula is not as straightforward as with our previous SPIN-based proposal where the properties to be verified where directly represented by LTL formulas (LTL is just the input language of SPIN for specifying the properties to be verified). Thus, to verify properties in Formula, in particular, the *selected existential* properties listed previously, we need to provide with a proposal to specify such CTL constructors in Formula. We would like to note that in our proposal we have considered that each proposition variable in a CTL formula (such as `p`, `q`, etc) refers to states. For example, the formula `AFp` would refer to the fact that "the state `p` will always eventually happen, no matter what execution path is taken". For this reason, from now on, when using the CTL formulas of the chosen patterns, we consider each proposition variable appearing in such formulas to be a state of the concrete guideline's state machine (that is, a state an object can go through during its lifetime). Next, we provide the main ideas behind the specification in Formula of the *selected existential* properties, and go on to describe our concrete proposal.

7.1. Ideas behind our proposal

Taking into account the Formula structural elements, we use queries to represent CTL formulas. Thus, the key idea behind our proposal is based on the semantics of Formula queries, both *existential quantification* and *universal quantification*. The chosen quantification strategy would allow us to represent a concrete type of property. More specifically, based on our proposal for the Formula specification of guidelines, we would like to make the following remarks leaning on Table 7:

- *Existential quantification*. It refers to the *natural* semantics of queries, that is, a query is true iff the right hand side of the definition is true for some substitution. We can use it to prove the existence of a guideline trace where a specific fact is hold. Checking an *existential quantification* query can result in two different results: (1) a *concrete trace* verifying the existence of a specific fact (that is, the trace

Table 7
Use of Formula queries in our proposal.

Quantification	Strategy	Queries represent	Looking for	Desired result
Existence	Optimistic	Valid states	Success	A trace
Universal	Pessimistic	Forbidden states	Failure	Unsatisfiable

includes a concrete state), or (2) *unsatisfiable*, if it is not possible to find a trace that satisfies such a specific fact. When using this quantification type, we say that we follow an *optimistic* strategy, since the query represents a *valid* situation (the desired specific fact), looking for *success* (a *concrete trace*).

- *Universal quantification*. We can use it to prove that a specific fact is verified in all traces of a guideline. As advanced previously, the *universal quantification* in Formula can be achieved by verifying the negation of a query representing the opposite of the original fact. In this case, the results are just the opposite than in the previous case: (1) *unsatisfiable*, if all traces in the guideline verify the original fact, or (2) a *concrete trace* verifying the negation of the original fact, which would prove that the original fact (not negated) does not hold in at least one trace. When using this quantification type, we say that we follow a *pessimistic* strategy, since the query represents a *forbidden* situation (the negation of the original fact), looking for failure (*unsatisfiable*).

Regarding the definition of the Formula *queries*, since the CTL formulas refer to system executions (that is, guideline's execution traces or, in other words, sequences of state configurations), taking into account our proposal for the Formula specification of guidelines, we represent such traces basically in terms of the `StateConfiguration` data type (defined in the *InstanceLevel* domain). Thus, the queries are defined also in such a model. A generic state configuration that takes place in a concrete moment `t` would be represented as `StateConfiguration(t,_,_)`, where each of the remaining two elements in the definition of the `StateConfiguration` data type are fresh variables (represented by the unknown _ symbol).

Taking this into account, next, we describe the formalization of the *selected existential* properties in Formula. In particular, we present in detail the final Formula specification of such properties. These Formula specifications use generic predicates in their definition (such as `p` or `q`) which would just need to be substituted by the desired fact (a *state* as noted previously). Although the formalization in Formula of these properties has been obtained after performing a more complex process, here we do not delve into detail about such a process but we describe the final Formula statements (from now on *Formula Property specification patterns*). This is mainly because the final user only needs to apply such Formula patterns to the concrete Property desired to be verified. We refer to the supplementary material [80] for a complete description of such a process.

7.2. Translation to Formula of the selected existential properties

Translations of *selected existential* properties from CTL to Formula Property specification patterns are presented in Tables 8 and 9. Next, we describe them in detail, illustrating our explanations by means of the verification of concrete properties (*example properties*) defined within the *AP guideline* context.

Everywhere eventually. Since this Property refers to a CTL *universal* modality (for all paths `A`, eventually `F`), we have adopted a *universal-pessimistic* strategy, defining a query `q1` that represents the negation of the fact given by `p` (that is, a *forbidden state*), aimed at obtaining *unsatisfiable* as result (see Table 7). More specifically, as presented in Table 8, the defined query aims at finding a sequence of state configurations where the state represented by `p` is never reached (note the use of the Formula term `fail`). Finally, such a query is directly included in

Table 8
Translation of the *Everywhere eventually* and *Always eventually* patterns.

Everywhere eventually

Description: The state `p` will always eventually happen, no matter what execution path is taken.

CTL: `AFp`

Formula:
```
q1:=fail StateConfiguration(t,StateInstance(p,_),_),
  Bound(end), t < end.
conforms:= ...& q1.
```

Always eventually

Description: No matter where in the system execution we are, the state `p` will always eventually happen.

CTL: `AG(AFp)`

Formula:
```
q1:= StateConfiguration(t,_,_),
  fail StateConfiguration(tn,StateInstance(p,_),_),
  Bound(end), t < tn, tn < end.
conforms:= ...& q1.
```

the `conforms` query. Given the semantics of Formula *queries*, the verification of this query in Formula would result in (1) *unsatisfiable*, which means that there is no guideline's application which does not go through the state given by `p` (which would prove the `AFp` property), or (2) a sequence of state configurations representing a guideline's application where the object go through the state given by `p` (which would prove that the Property is not true). This translation is illustrated by the following example property.

Example Property 1. This Property formalizes the restriction "no matter what guideline's application path is taken, the aliquot is always eventually placed into racks and registered (state *InRackAndBeingRegisteredS4*)". More specifically, the application of our pattern in Table 8 basically consists of defining a query representing the fact that the state *InRackAndBeingRegisteredS4* is never possible. Thus, Formula would try to generate a sequence of state configurations that does not include the state *InRackAndBeingRegisteredS4*. Finally, the verification of this Property in Formula would result in the execution trace labeled as *(a)* in Fig. 2, showing that the Property is not satisfied by the *AP guideline*.

Example Property 1

CTL: `AF InRackAndBeingRegisteredS4`

Formula:
```
q1:= fail StateConfiguration(t,StateInstance("InRackAnd-
  BeingRegisteredS4",_),_),
        Bound(end),t < end.
    conforms:= ...& q1.
```

At this point, a special remark could be made regarding the use of the *optimistic* strategy for the representation in Formula of CTL properties. As an example, although not being a *selected existential* property, we will use the *Possible existence* pattern that refers to properties directly related to the existence of at least one path in the system execution in which some condition must hold. More specifically, this pattern (represented in CTL as `EFp`) refers to that "it is possible for `p` to happen, that is, there is a path (`E`), along which eventually (`F`) `p` occurs". We would like to note that, in contrast to `AFp`, to prove this Property we need to represent the existence of a path along which the state `p` eventually occurs, but there could be a path where the state `p` does not occur. Taking this into account, for the specification in Formula of this pattern we follow an *optimistic* strategy and consider the *existential* quantification characteristic of queries, trying to find a sequence of state configurations where it is possible for the state `p` to take place. More specifically, the representation in Formula of this pattern results in the definition of the following query, included in the `conforms` one: `q1:= StateConfiguration(t,StateInstance`

Table 9
Translation of the *Liveness* patterns.

Liveness reachability		
	Description:	At any time during the execution of the system, the state `p` will eventually become possible.
	CTL:	`AG(EFp)`

	Formula:	`q1:= StateConfiguration(t,_,_),`
		`fail StateConfiguration(tn,StateInstance(p,_),_),`
		`Bound(end), t < tn, tn < end.`
		`q2:= StateConfiguration(t,src,_), src.name!= "Final",`
		`fail StateConfiguration(tm,PseudostateInstance(_,Pseudostate(_,choice)),_),`
		`Bound(end), t < tm, tm < end.`
		`conforms:= ...& q1 & q2.`

Temporal Implication		
	Description:	Whenever the state `q` holds, there exists an execution path where the state `p` will eventually be reached.
	CTL:	`AG(q →EFp)`

	Formula:	`q1:= StateConfiguration(t,StateInstance(q,_),_),`
		`fail StateConfiguration(tn,StateInstance(p,_),_),`
		`Bound(end), t < tn, tn < end.`
		`q2:= StateConfiguration(t,StateInstance(q,_),_), q!="Final",`
		`fail StateConfiguration(tm,PseudostateInstance(_,Pseudostate(_,choice)),_),`
		`Bound(end), t < tm, tm < end.`
		`conforms:= ...& q1 & q2.`

`(p,_),_), Bound(end), t < end`. If the verification of this query results in a *concrete trace*, it would correspond to a sequence of state configurations where the object achieves the state `p`. If the result is *unsatisfiable*, the formula will be false. Later, in Section 8, we show the verification of a *Possible existence* Property applied to our case study.

Always eventually. This pattern, represented in CTL as AG(AFp), could also be read as "for all computations (`A`), and for all states in it (`G`), from all paths from that state (`A`), eventually (`F`) `p` is true". Again, since this Property follows a CTL *universal* modality, for its translation to Formula we have followed similar ideas than with the *Everywhere eventually* pattern (see Table 8). In particular, in this case, if the verification of the defined query `q1` results in a specific trace, then the Property would be false since such a trace would correspond to a sequence of state configurations where `p` never happens (`fail StateConfiguration(tn,StateInstance(p,_),_)`). If Formula shows that the model is *unsatisfiable*, the Property would have been proven.

Liveness reachability. This pattern (represented in CTL as `AG(EFp)`) states that "for all computations (`A`), and for all states in it (`G`), there is a path (`E`) along which eventually (`F`) `p` occurs". We will describe the main ideas of the translation of this pattern by using the following Property (from now on *Example Property 2*).

Example Property 2. This Property aims at verifying the fact that "no matter where in the guideline's application process we are, the cancellation of the aliquoting process will eventually become possible", that is, "from any state during the application of the guideline to an aliquot, it is eventually possible for the aliquot to get the state *CancelledS6*."

Example Property 2

CTL: `AG(EF CancelledS6)`

Formula:
```
q1:= StateConfiguration(t,_,_),
     fail StateConfiguration(tn,StateInstance("Cancell-
edS6",_),_),
     Bound(end), t < tn, tn < end.
q2:= StateConfiguration(t,src,_), src.name!= Final,
     fail StateConfiguration(tm,PseudostateInstance(_,P-
seudostate(_,choice)),_),
     Bound(end), t < tm, tm < end.
conforms:= ...& q1 & q2.
```

For the representation in Formula of this pattern, we have followed a *universal-pessimistic* strategy which consists of trying to find a sequence of state configurations which verifies the negation of the Property `AG(EFp)`, that is, a sequence of state configurations where is not possible for `p` to take place. Thus, the lack of such a sequence would show that the Property is *true* for all sequence of state configurations. For such a task, we represent in Formula the negation of the property, and later, we use Formula to check it on the model.

More specifically, first, we define a query `q1` ((see Table 8)) which states that "at any time during the system's execution (`StateConfiguration(t,_,_)`), the object never achieves the `p` state (`fail stateConfiguration(tn,StateInstance(p,_),_)`). That is, this query represents a sequence of state configurations (traces) where `p` does not take place (note the term `fail` in the definition of `q1`). By verifying such a query in the `conforms` statement, we aim at obtaining traces where `p` does not become possible.

Example Property 2. By verifying query `q1` defined for this property, we could get sequences of configurations such as (1) the one containing state *ReadyToBeAliquotedS1* to the final state (execution trace labeled as *(a)* in Fig. 2) (from now on sequence *seq1*), or (2) a sequence containing the state *AliquotedS5*, which also ends in a final state, without entering the state *CancelledS6* (for example, the execution trace labeled as *(c)* in Fig. 2) (from now on *seq2*). It is worth noting that both sequence of configurations are different since *seq1* contains the state *ReadyToBeAliquotedS1* from which, if taken transition *t1b*, it could be possible to reach the state *CancelledS6*, because of the *choice* pseudostate. However, *seq2* contains state *AliquotedS5*, but from this state is not possible to achieve the state *CancelledS6*.

Since the verification of this query could lead to two different types of traces, it is required to dismiss those which correspond to traces containing states which, taken other paths in the system's execution, could lead to the desired state `p` (such as sequences of type *seq1* in our example). For such a task, second, we define another query `q2` which aims at resulting in sequences of configurations with states which do not lead to choice pseudostates. Finally, by verifying both `q1` and `q2` (see Table 9) we aim at finding any trace which both does not end the required state `p`, and which have an state from which it is not possible to reach the required state by taking any other path. The existence of such a trace would demonstrate that the Property `AG(EFp)` is not satisfied by the model, unlike obtaining an *unsatisfiable* model which would show that the Property is *true*.

Example Property 2. The result of verifying the query `q2`, defined for this property, would result in the trace labeled as *(c)* in Fig. 2.

Taking into account the verification of `q1` and `q2` in the `conforms` query, we obtain the same trace, which means that there is at least a state during the guideline's application where it is not possible for the aliquot to reach the *CancelledS6* state (at any time during the execution of the guideline, it is not possible for the aliquoting process to be cancelled). Thus, proving that the Property is not satisfied.

Temporal implication. Taking into account the translation of the *Liveness reachability* pattern, the specification in Formula of the *Temporal implication* pattern (`AG(q →EFp)`) results almost straightforward by substituting, in both queries of the translation of *Liveness reachability* (`q1` and `q2`), the undetermined state configuration (that is, the second parameter in (`StateConfiguration(t,_,_)` and `StateConfiguration(t,src,_)`, respectively) by `StateInstance(q,_)`.

8. Verification of the *AP guideline*. Example of application

As an example, in this section we illustrate the usefulness and effectiveness of our Formula-based approach by applying it to our particular case study (the translation and verification of the two *AP guideline*'s requirements shown in the previous section could also be considered as part of this evaluation). Here, we present the verification process to be followed when using our proposal, together with the results obtained from verifying different types of requirements in the *AP guideline*. More specifically, during the verification process of the *AP guideline* we have proven not only requirements not being able to check using our previous proposal based on SPIN (i.e., a *Liveness property*), but also several types of requirements we considered useful to be verified in guidelines in general. Next, we describe our experiences and lessons learned.

Step 1. We have started from the state machine model of the *AP guideline* of Fig. 2. We have created it by using the UML2 Eclipse plug-in [78], obtaining a `.uml2` extension file. Having the resulting state machine model, we have firstly flattened it and then translated it into the input specification language of the formula tool using our Eclipse plug-in. Given the size of the state machine, with a small number of possible transitions to be executed, we have chosen `10` as value of the `n` parameter in the `Bound` term. As a result, we have obtained a `.4ml` extension file with 153 lines.

Steps 2 and 3. The *verification concerns* we have proven on the guideline are organized based on the three main scenarios described for Formula: *Consistency checking*, *Synthesis*, and *Design space exploration* (the previous two properties would match the *Synthesis* scenario).

- *Consistency checking*. Thanks to the verification of the `done` query included in our *InstanceLevel* domain, our proposal method allows us to make sure that the defined guideline specification is satisfiable, that is, that exists at least a valid execution trace that reaches a final state at some point. More specifically, there is a sequence of state configurations an aliquot can go through during its life as the guideline is applied ending in a final state. After executing Formula with the specification file resulting from the previous step, it outputs a chain of `StateConfiguration` facts referring to a finished guideline's application reaching a final state. More specifically, Formula returns the execution trace labeled (a) in Fig. 2. We note that this `done` query is always included in the `conforms` query of the *InstanceLevel* domain to ensure that Formula returns valid traces. We also want to note that defining a query similar to `done` with any configuration state other than the final one would result, if exists, in an execution trace reaching such a state just once.
- *Synthesis*: here we show the verification of several *verification concerns* that match with this scenario (that is, we have proven the existence or not of valid execution traces verifying a certain *concern* on a guideline). Next, we show three different situations: (1) checking a *requirement* that could not be verified with our previous proposal [6], (2) checking a type of Property we verified in [6] by using our previous proposal to show that there are situations to be checked by using both proposals, and (3) proving whether there exists a guideline application which complies with specific *customized partial Formula statements*.

1. One of our main goals was to address the shortcomings of our previous, in particular, by checking commonly used properties such as the ones included in the *Liveness* pattern (in particular, the *Liveness reachability* subpattern, since neither this formula nor its negation are represented in LTL). In the previous section, we have described the Formula translation of *Example Property 2* `AG(EF CancelledS6)`, aimed at verifying the fact that at any time during the guideline to an aliquot, the cancellation of the aliquoting process (represented by the state *CancelledS6*) will eventually become possible. We have verified this property, showing that it is not satisfied.
2. In the second case, we show the verification of a Property of type *Possible existence*, which aims to verify whether "it is possible for something to happen, that is, a Property may hold in some paths but not all the paths of the execution [6]." This type of Property can not be directly represented in the LTL language, but SPIN can be used to check them by means of verifying the negation of the LTL property. In particular, in [6] we showed the verification of a *Possible existence* Property which required us to perform several steps (defining in LTL the negation of the property, generating the corresponding "never claim" formula [8], using SPIN to prove such a formula, and interpreting the resulting counterexample showing the existence of at least one guideline application in which the positive Property is hold). Here, we show how we can also verify this type of properties in an easy way by using our Formula-based proposal. In particular, we have proven whether, given an aliquot in a specific state (*ReadyToBeAliquotedS1*) there exists at least a guideline application which leads to a specific aliquot's state (*AliquotedS5*). We have defined the Formula query `q` representing this Property (again, we have included the defined query in the `conforms` query of the *InstanceLevel* domain):

```
q:= StateConfiguration(t,S_S1,_),
    StateConfiguration(tn,S_S5,_),
    Bound(end),t < tn, tn < end.
conforms:= ...& q.
```

Formula has returned the trace labeled (c) in Fig. 2, representing a guideline's execution where, having been the aliquot in the state *ReadyToBeAliquotedS1*, the aliquot's state leads to *AliquotedS5*. We note that we can also prove this Property obtaining similar results, by using the following formula based on the `fail` Formula element.

```
q':= StateConfiguration(t,S_S1,_),
    fail StateConfiguration(tn,S_S5,_),
    Bound(end), t < tn, tn < end.
conforms:= ...&!q'.
```

3. In the third case, we have simulated the situation in which a sample is being aliquoted, but the guideline's application is not yet finished, and thus, we have a portion of an execution trace (for example, the trace from the initial state to the *InFillingProcessS2* state). Taking this into account, we have proven if, considering such a guideline's application, it is possible to successfully finish the aliquoting process. More specifically, we

have included in the *Execution* partial model the corresponding `Trigger` partial statements, including the transition instances conforming to the portion of the trace:

```
Trigger(0, t0)
Trigger(1, t1)
Trigger(2, t1b)
```

The previous statements mean that, if `t0` has taken place, the aliquot has gone from the initial state to the *ReadyToBeAliquotedS1* state, while if `t1` and `t1b` have taken place, the aliquot has arrived the *InFillingProcessS2* state). We have also included in the *InstanceLevel* domain the following query to force Formula to return a trace where the aliquoting process is successfully finished (*AliquotedS5*):

```
q:= StateConfiguration(t,S_S5,_),
   Bound(end),t < end.
conforms:= ...& q.
```

After executing Formula, the tool has searched a execution trace which includes not only the two state configurations corresponding to the stay in both states (that is, *ReadyToBeAliquotedS1* and *InFillingProcessS2*), but also a set of subsequent configuration states referring to valid states until achieving *AliquotedS5*. Thus, the results show that there exists a possible set of states reaching the final aliquoting of the sample (in particular Formula shows again the trace labeled as (c) in Fig. 2).

- *Design space exploration*. The guideline's Formula specification can be used to inspect the clinical guideline represented by the state machine, to search for conforming execution traces and choose those which satisfy the desired *verification concerns*. We note that our previous proposal based on the SPIN model checker did not provide us with this possibility. As an example of this scenario, we have used Formula to obtain several execution traces referring to different guideline's application achieving the final state (see the *Consistency checking* point). More specifically, we have started from the main Formula file, without including any additional verification concern (just `done`), and used Formula to explore the model's design space, returning, in particular, the execution traces depicted in the *Instance level* of Fig. 2.

9. Discussion

There are several strengths regarding our proposal that we want to highlight. In this paper, we have presented a formal framework complementary to our previous one in [6] for verifying clinical guidelines, which mainly addresses the shortcomings of our previous approach. As a result, both proposals constitute an overall framework where their complementary use would provide with a more complete verification of guidelines.

On one hand, the framework presented here shares characteristics with our previous one in [6]. First, the only two manual steps that must be performed in both proposals correspond to (1) the representation of the clinical guideline as a UML state machine (step guided by the representation patterns presented in [6]), and (2) the specification of the *verification concerns* to be checked in the guideline. The remaining steps are performed automatically in both proposals by using MDD-based techniques, which makes the process faster and less error-prone. Second, the extension we provide to our previous Property specification patterns is, as the hierarchy of patterns provided in [6], general enough to be used to specify requirements in contexts other than the clinical one. Finally, we also provide with a tool (implemented in form of an Eclipse plug-in) by means of which the Formula model is automatically generated by only selecting the menu item the plug-in provides. Thus, it allows the verification process to be easily adapted to changes in the definition of a guideline that was previously verified with our proposal, since it will only be necessary to manually modify the state machine which represents the guideline (and therefore the properties defined from it), being the Formula model automatically generated.

As shown in the previous section, we have demonstrated the feasibility of our proposal by applying it to a guideline chosen for its simplicity, checking on it different types of properties which can provide useful information regarding everyday guideline applications. Additionally, our framework has been applied to other real-life guidelines aimed at covering other contexts within the medical care system, and which were used in our previous work in [6], obtaining also encouraging results. Such applications include a guideline utilized for the management of infections related to intravenous catheters (IRC) (used in a Spanish hospital and developed on the basis of a guideline published by the US Agency for Health Care Research (AHCR) and Quality National Guideline Clearinghouse (NGC)), and a guideline for the management of obesity in primary care, also published by the NGC.

On the other hand, although we have applied our verification approach to different guidelines obtaining encouraging results, we have to recognize that there are certain limitations to the presented work. Here, we touch upon several of these issues. Firstly, as described previously, the specification of patterns in Formula is not as straightforward as in our previous proposal since it requires to translate the CTL-based representation of the chosen pattern into Formula (while with SPIN we can use the LTL rule provided by the pattern). However, we think that this shortcoming is somewhat attenuated by our proposal for the translation of our Property patterns to Formula (in particular, *selected existential* properties, which have mainly motivated this work), which help non-experts in the specification language of the Formula tool to easily write the formal specifications of such properties (thus easing the verification process). Second, our approach for the translation of guidelines represented by state machines into the Formula language specification does not currently support several UML State Machine elements (such as guard variables, or pseudostates other than *initial* and *choice*). More specifically, our proposal allows us to register the guard condition satisfied when a choice element in the state machine is found, but does not allow us to represent the logical expressions constituting such guards and, thus, what variable values make the guard condition satisfied. For this reason, we can not define requirements based on such variables. Therefore, in the near future we plan to extend our approach in order to support such elements. Third, among the guideline categorization provided by the US Agency for Health Care Research (AHCR) [81], the guidelines used for showing the feasibility of our proposal cover a wide range such as management, diagnosis, treatment or prevention, but the application of our proposal the other categories (such as counselling) constitutes an interesting line of further work.

10. Related work

Since the work we have presented in this paper covers different though related topics, next we compare our proposal to relevant related work regarding two criteria related to (1) techniques for the verification of clinical guidelines, focusing mainly on those based on CLP, and (2) verification of systems based on model finders.

10.1. Techniques for the verification of clinical guidelines

Validation and verification play a key step in clinical guidelines' life-cycle since they contribute to decreasing errors and increasing the quality and safety of both clinical guidelines and their implementation into useful decision support systems. Although the terms validation and verification are not clearly differentiated when referring to clinical guidelines, we adopt the definitions considered by Peleg in [31]. More specifically, *validation* refers to a process which aims at establishing that

the guideline's requirements are captured in its specification. In contrast, *verification* concerns mathematical proof that an implementation meets its formal specification. Additionally, we would like to include *critiquing* that, although it is not directly related to validation and verification, it employs similar methods at the other two approaches and therefore it is interesting for the presented work. More specifically, *critiquing* aims at comparing actual clinical actions performed by a physician with a predefined "ideal" set of actions suggested by a guideline in order to identify various types of non-compliance (for example, actions unsupported by patient findings, conflicting actions or missing mandatory actions) [32,35]. Thus, it is used when a guideline is being applied to a patient.

Validation and verification. In [31] Peleg presents a methodological review of Computer-interpretable clinical guidelines, providing, in particular, an overview of different techniques of *validation*, which include inspection, testing, and *verification*, where three categories of techniques are identified; the first and second categories are related to the verification of a guideline, while the third one concerns checking inconsistencies between guidelines. More specifically, the first category deals with proving that the guideline specification is internally consistent and free of anomalies. The second category aims at proving that the guideline specification satisfies a set of desired properties. Finally, the third category focuses on identifying conflicts in pairs of guidelines.

Our proposal could be considered to be within the two first verification categories since it allows both the detection of anomalies within the guideline specification and the verification of satisfying properties. There are several works which have been published in the literature concerning the verification of guidelines since our previous proposal. Among those which use CLP as ours, we cite the work by Mather et al. [36] as the most similar to our proposal. In [36] authors present a work, still under development, which consists of a clinical decision support system called *ATTENTION* for synthesizing and managing longitudinal treatment plans. More specifically, *ATTENTION* aims to combine state-of-the-art formal modeling and constraint solving with clinical information systems to synthesize complex cancer treatment plans that are also executable. Similarly to our work, they aim to use Formula, and thus CLP, to represent the guideline's models so that Formula facilitates (1) checking the well-formedness of guideline models, (2) representing their execution semantics, (3) generating executable (patient specific) guideline instances, and execution traces and (4) verification of invariants. One of the main drawbacks of that work is that, although it seems promising, several of the components they describe are presented to be still under development (and it seems that there have not been current publications of the work). But another shortcoming that distinguishes this work from ours is the fact that they represent each state in an over simplified way by means of three elements: a drug ontology, a set of patient records and pharmacy data. From our point of view, this way of representation is not suitable to be applied to any kind of state included in a guideline (for example, those states that can not be represented by concrete values of such three elements). Our proposal for personalization and customization of states, assigning a name per each state (as we established in our patterns to represent guidelines as state machines [13]), can be used to represent any kind of guideline's state. Although, we could also introduce an associative memory or repository that would provide relevant data for a given state in our proposal, we have not considered such a richer description of states as it is not necessary for verification.

Other proposal which uses CLP for guidelines' verification is the extensive work presented by Wilk et al. [32–34], which uses CLP to identify conflicts in pairs of guidelines (thus this work is included in the third category identified by Peleg [31]). More specifically, CLP is used to mitigate adverse interactions that could occur when two concurrency guidelines are applied to a patient with comorbid diseases. Although having a different goal than ours, both share several similarities. Their mitigation algorithm starts with each CPG being transformed by performing several steps into a set of the constraints that make up a constraint programing (CP) model. Subsequently, the individual CP models are solved in such a way that they define a solution as a set of variable/value pairs satisfying the constraints. Finally, the solutions are parsed for potential contradictions looking for a feasible combined therapy. As CP system they use the open source tool ECLiPSe for solving the CP models of the guidelines [34].

Critiquing. Several critiquing proposals have been presented in the literature. An example is [35,82], where the authors use model checking to check guidelines, represented as state transition systems, against a set of temporal logic formulas which describe actual actions derived from real world patient data. The main difference between our approach an these works is that, since they are based on critiquing, they focus mainly on comparing clinical actions performed by a physician (thus, using available patient data as input information) with a predefined set of actions as described by a clinical guideline. Our second scenario *synthesis*, and in particular, the use of *customized partial Formula statements*, could be considered similar to this work.

10.2. Systems verification based on model finders

Our work relies on the use of the Formula tool for guidelines' verification. Regarding the use of Formula instead of other analyzers, in particular, the Formula authors presented in [65] a comparison with other tools, both SAT (Boolean Satisfiability) solvers and alternatives such as *ECLiPSE* and *UMLtoCSP*, focusing mainly on *Alloy* [83–85], since it is the closest tool to Formula. Although the Formula authors provided a careful comparison with *Alloy*, it is worth highlighting the strengths of Formula, in contrast to *Alloy*, which presents distinctive strengths compared to other similar tools, including better expressiveness. We do not delve into more details regarding such a comparison but a description can be seen in [65].

11. Conclusion

In this paper we present a revision and extension of the proposal we gave in [6]. The most significant contributions of this research paper are the following. First, we provide a more complete set of patterns for defining commonly occurring types of requirements in guidelines. Second, we give support for the verification of a wider range of patterns, by combining the use of our previous proposal based on the SPIN model checker, with the Formula model finder. In particular, our proposal based on the use of Formula mainly addresses the shortcomings of our previous approach, while providing additional verification functionalities. Particularly, we have presented a guideline's state machine to Formula transformation proposal and defined an Eclipse plug-in based on MDD for automatically generating the comprehensive Formula specification required by the Formula model finder to carry out the verification process on the guideline. Our proposal also allows the verification of properties conforming to the *selected existential* patterns, and supports translation of such patterns to Formula, thus helping non experts in their formal specification. Depending on the requirement to be verified, the user can chose between using either the SPIN model checker proposal or the Formula model finder approach. We want also to note that, although our proposal is presented specifically to be used with guidelines, it can be applied to other systems in other contexts, provided that the system is represented as a state machine.

Conflict of interest

The author declared that there is no conflict of interest.

Acknowledgements

This research was funded by the Spanish Ministry of Economy and Competitiveness, project number EDU2016-79838-P.

Appendix A. Main ideas regarding the use of property specification patterns

Next, we provide an overview of the key ideas regarding the use of Property specification patterns as stated by the Dwyer et al. in [18,86,52]. A *Property specification pattern* is referred to as "a generalized description of a commonly occurring requirement on the permissible state/event sequences in a finite-state model of a system" [86]. The idea behind a Property specification pattern is that it "describes the essential structure of some aspect of a system's behavior and provides expressions of this behavior in a range of common formalisms" [52]. Each pattern "should be described in a form that can be understood by practitioners so that they can identify similar requirements in their systems, select patterns that address those requirements, and instantiate solutions that embody those patterns" [86].

In this paper, we start from the hierarchy of patterns we previously established in [6], and which has been extended by the set of patterns and subpatterns included in the revision presented in this paper (which is described in Section 3). On the one hand, the concrete LTL and CTL formulas of the patterns obtained from Dwyer et al. and from Ryndina et al. can be consulted in [18,52,20] respectively. On the other hand, in [6] the reader can find the formulas corresponding to the new patterns we proposed previously, while the formulas of the patterns and subpatterns provided in the presented work can be seen in Section 3. We would like to note that, independently of the source (Dwyer et al., Ryndina et al., or our patterns), to make our patterns easier to use, all the patterns come with descriptions that aim at illustrating HOW to map well-understood, but imprecise, conceptions of system behavior [52] (guidelines' application in our case) formulated by non-experts in a specification language, INTO precise statements in common formal specification languages [52] (such as LTL and CTL, used in our proposal).

How to find the pattern for the Property desired to be verified. To help the user to apply the patterns, they are organized into a simple hierarchy in terms of the different kinds of system behaviors they describe (distinguishing mainly between *occurrence* and *order*), and with links between related patterns (see Fig. 7). As suggested in [52], a non-expert should be able to locate the suitable pattern (if not the desired pattern, at least one close to what she/he wants) by following easy steps:

1. The user starts with the formulation, in natural language, of the desired Property to be verified in the guideline.
2. Second, the user searches down the hierarchy to the kind of Property she/he needs. In particular, she/he can look for some specific patterns that sound relevant. For such a task, the user can follow the overall Property specification patterns presented in Fig. 7 (which includes the overall hierarchy of patterns proposed in this paper). More specifically, the user can consult the description (showing the patterns' intention) of the overall patterns in [52] (where Dwyer et al. patterns are presented), at [20] (where Ryndina et al. patterns are described), and finally in [6] and in this paper (where we include a description of our proposed patterns). The user can follow links to related patterns, until the desired pattern, that is, the pattern whose description better matches the desired Property formulation.
3. When the user finds the desired pattern, she/he can base on the mappings into the concrete formalisms (LTL, CTL) the pattern provides, so that she/he can formulate an instance of the pattern which refers to her/his desired property.
4. Finally, depending on the temporal logic, the user can utilize a concrete verification tool to finally check the Property in the guideline. For example, if the Property matches a pattern able to be represented in LTL formula, the user can utilize our SPIN-based proposal in [6], while if the Property matches any of the *selected existential* patterns, she/he can use the Formula-based proposal presented in this paper.

As a way of example, let's follow each of the previous steps by considering a concrete Property within the case study of the *AP guideline*, in particular, *Property 1* described in Section 7.

1. First, let's suppose that the user has the intention to verify that "no matter what guideline's application path is taken, the aliquot is always eventually placed into racks and registered (state *InRackAndBeingRegisteredS4*)". This Property can equivalently be read as "for all guideline's application paths, the aliquot is eventually in *InRackAndBeingRegisteredS4*".
2. Taking into account the hierarchy of patterns, this Property clearly corresponds to an *Occurrence* pattern. Thus, we search down the hierarchy of patterns looking for these which sound relevant (we can rely on Fig. 7 to identify the source where we can find the description of each pattern, that is, either Dwyer et al. [52], Ryndina et al. [20], etc.). More specifically, since the Property refers to the reachability (or the *existence*) of certain system states, we focus on such a pattern. Among its subpatterns, we can dismiss all but the *everywhere existence*, chosen as our desired pattern.
3. As shown in Fig. 7, this pattern was proposed by Ryndina et al. [20]. Additionally, it provides a mapping to CTL (it is not supported in LTL) as `AFp`. Thus, the formulation of our *Property 1* into CTL would be `AF InRackAndBeingRegisteredS4`.
4. Finally, as described in Section 7, we can use our proposal to obtain the Formula expression corresponding to such a property. We just need to substitute *p* by `InRackAndBeingRegisteredS4` in the *Formula specification pattern* of Table 8. Finally, the resulting Formula expression can be included in the guideline's Formula specification to check whether the guideline verifies it.

References

[1] Institute of Medicine, Guidelines for Clinical Practice: from Development to Use, National Academy Press, Washington, D.C., 1992.
[2] C. Papadopoulos, The development of Canadian clinical practice guidelines: a literature review and synthesis of findings, J. Can. Chiropr. Assoc. 47 (1) (2003) 39–57.
[3] L. Giordano, P. Terenziani, A. Bottrighi, S. Montani, L. Donzella, Model checking for clinical guidelines: an agent-based approach, AMIA Annual Symposium Proceedings, 2006, pp. 289–293.
[4] A. ten Teije, M. Marcos, M. Balser, J. van Croonenborg, C. Duelli, F. van Harmelen, P.J.F. Lucas, S. Miksch, W. Reif, K. Rosenbrand, A. Seyfang, Improving medical protocols by formal methods, Artif. Intell. Med. 36 (3) (2006) 193–209.
[5] National Health and Medical Research Council, A guide to the development, implementation and evaluation of clinical practice guidelines, in: Canberra: NHMRC, 1998. Available at: < https://www.health.qld.gov.au/_data/assets/pdf_file/0029/143696/nhmrc_clinprgde.pdf > .
[6] B. Pérez, I. Porres, Authoring and verification of clinical guidelines: a model driven approach, J. Biomed. Inform. 43 (4) (2010) 520–536.
[7] OMG, UML 2.4.1 Superstructure Specification, document formal/2011-08-06, August, 2012. Available at: < http://www.omg.org/ > (last visited on February 2019).
[8] SPIN and PROMELA reference manual, website: < http://spinroot.com/spin/whatispin.html > (last visited on February 2019).
[9] J. Joxan, M.J. Maher, Constraint logic programming: a survey, J. Logic Program. 19/20 (1994) 503–581.
[10] FORMULA – Modeling Foundations, website: < http://research.microsoft.com/en-us/projects/formula/ > (last visited on February 2019).
[11] J.A. Casasnovas, V. Alcaide, F. Civeira, E. Guallar, B. Ibañez, J.J. Borreguero, M. Laclaustra, M. León, J.L. Peñalvo, J.M. Ordovás, M. Pocovi, G. Sanz, V. Fuster, Aragon workers' health study – design and cohort description, BMC Cardiovasc. Disord. 12 (1) (2012).
[12] E. Domínguez, B. Pérez, A.L. Rubio, M.A. Zapata, J. Lavilla, A. Allué, Occurrence-oriented design strategy for developing business process monitoring systems, IEEE Trans. Knowl. Data Eng. 26 (7) (2014) 1749–1762.
[13] I. Porres, E. Domínguez, B. Pérez, A. Rodríguez, M.A. Zapata, A model driven approach to automate the implementation of clinical guidelines in decision support systems, in: Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'08), 2008, pp. 210–218.
[14] S. Sen, B. Baudry, D. Precup, Partial model completion in model driven engineering using constraint logic programming, in: Proceedings of the International Conference on Applications of Declarative Programming and Knowledge Management (INAP'07), 2007.
[15] J. Bézivin, Model driven engineering: an emerging technical space, International Summer School on the Generative and Transformational Techniques in Software

Engineering (GTTSE'05), Springer-Verlag, 2006, pp. 36–64.
[16] OMG, Meta Object Facility Specification, VERSION 2.5.1, formal/16-11-01, 2016. Available at: < http://www.omg.org/ > (last visited on February 2019).
[17] R.L. Cobleigh, G.S. Avrunin, L.A. Clarke, User guidance for creating precise and accessible property specifications, Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT/FSE'06), ACM, 2006, pp. 208–218.
[18] M.B. Dwyer, G.S. Avrunin, J.C. Corbett, Patterns in property specifications for finite-state verification, Proceedings of the 21st International Conference on Software Engineering (ICSE'99), IEEE, 1999, pp. 411–420.
[19] J. Yu, T. Manh, J. Han, Y. Jin, Y. Han, J. Wang, Pattern based property specification and verification for service composition, Lecture Notes in Computer Science vol. 4255, Springer, 2006.
[20] K. Ryndina, Improving Requirements Engineering: An Enhanced Requirements Modelling and Analysis Method (Ph.D. thesis), Department of Computer Science, University of Cape Town, 2005. Available at: < http://pubs.cs.uct.ac.za/archive/00000201/01/final_ryndina_thesis.pdf > (last visited on February 2019).
[21] K. Ryndina, P.S. Kritzinger, Analysis of structured use case models through model checking, South Afr. Comput. J. 35 (2005) 84–96.
[22] A. Pnueli, The temporal logic of programs, Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS'77), vol. 0, IEEE Computer Society, Los Alamitos, CA, USA, 1977, pp. 46–57.
[23] E.M. Clarke, E.A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, Logic of Programs, Lecture Notes in Computer Science, vol. 131, Springer, 1981, pp. 52–71.
[24] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Trans. Program. Lang. Syst. 8–2 (1986) 244–263.
[25] A. Hommersom, P. Groot, M. Balser, P.J.F. Lucas, Formal methods for verification of clinical practice guidelines, Computer-based Medical Guidelines and Protocols: A Primer and Current Trends, Studies in Health Technology and Informatics, vol. 139, IOS Press, 2008, pp. 63–80.
[26] A. Rutle, F. Rabbi, W. MacCaull, Y. Lamo, A user-friendly tool for model checking healthcare workflows, in: Proceedings of the Fourth International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN'13), 2013, pp. 317–326.
[27] A. Bottrighi, L. Giordano, G. Molino, S. Montani, P. Terenziani, M. Torchio, Adopting model checking techniques for clinical guidelines verification, Artif. Intell. Med. 48 (1) (2010) 1–19.
[28] F. Rahman, J.K.F. Bowles, Formal verification of CNL health recommendations, Proceedings of the Integrated Formal Methods (IFM'17), Lecture Notes in Computer Science, vol. 10510, Springer International Publishing, 2017, pp. 357–371.
[29] A. Simalatsar, W. You, V. Gotta, N. Widmer, G.D. Micheli, Representation of medical guidelines with a computer interpretable model, Int. J. Artif. Intell. Tools 23 (3) (2014).
[30] B. Kamsu-Foguem, G. Diallo, C. Foguem, Conceptual graph-based knowledge representation for supporting reasoning in African traditional medicine, Eng. Appl. Artif. Intell. 26 (4) (2013) 1348–1365.
[31] M. Peleg, Computer-interpretable clinical guidelines: a methodological review, J. Biomed. Inform. 46 (4) (2013) 744–763.
[32] S. Wilk, W. Michalowski, M. Michalowski, K. Farion, M.M. Hing, S. Mohapatra, Mitigation of adverse interactions in pairs of clinical practice guidelines using constraint logic programming, J. Biomed. Inform. 46 (2) (2013) 341–353.
[33] M. Michalowski, M.M. Hing, S. Wilk, W. Michalowski, K. Farion, A constraint logic programming approach to identifying inconsistencies in clinical practice guidelines for patients with comorbidity, Proceedings of the 13th Conference on Artificial Intelligence in Medicine (AIME'11), Lecture Notes in Computer Science, vol. 6747, Springer, 2011, pp. 296–301.
[34] S. Wilk, M. Michalowski, W. Michalowski, M. Hing, K. Farion, Reconciling pairs of concurrently used clinical practice guidelines using Constraint Logic Programming, AMIA Annual Symposium Proceedings, 2011, pp. 944–953.
[35] P. Groot, A. Hommersom, P. Lucas, R. Merk, A. ten Teije, F. van Harmelen, et al., Using model checking for critiquing based on clinical guidelines, Artif. Intell. Med. 46 (1) (2009) 19–36.
[36] J.L. Mathe, J. Sztipanovits, M. Levy, E.K. Jackson, W. Schulte, Cancer treatment planning: formal methods to the rescue, Proceedings of the 4rd International Workshop on Software Engineering in Health Care (SEHC'12), IEEE Computer Society, 2012, pp. 19–25.
[37] B. Moszkowski, A temporal logic for multilevel reasoning about hardware, Computer 18 (2) (1985) 10–19.
[38] E.M. Clarke, O. Grumberg, K. Hamaguchi, Another look at LTL model checking, Formal Methods Syst. Des. 10 (1) (1997) 47–71.
[39] J.I. Perna, C. George, Model checking RAISE specifications, Tech. rep., UNU-IIST, International Institute for Software Technology, 2006. Available at: < http://sedici.unlp.edu.ar/bitstream/handle/10915/22134/Documento_completo.pdf?sequence=1 > (last visited on February 2019).
[40] E.M. Clarke, I.A. Draghicescu, Expressibility results for linear-time and branching-time logics, Proceedings of the Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop (REX'88), Springer-Verlag, London, UK, 1988, pp. 428–437.
[41] O. Kupferman, M.Y. Vardi, Relating linear and branching model checking, Proceedings of the International Conference on Programming Concepts and Methods (PROCOMET'98), 1998, pp. 304–326.
[42] J.C. Campos, J. Machado, A specification patterns system for discrete event systems analysis, Int. J. Adv. Rob. Syst. 10 (8) (2013) 315.
[43] P. Drábik, Modular Verification of Biological Systems (Ph.D. thesis), Università di Pisa. Dipartimento di Informatica, 2011.
[44] E.A. Emerson, C. Lei, Modalities for model checking: branching time logic strikes back, Sci. Comput. Program. 8 (3) (1987) 275–306.
[45] D. Lehmann, A. Pnueli, J. Stavi, Impartiality, justice and fairness: the ethics of concurrent termination, Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP'81), 1981, pp. 264–277.
[46] M. Maidl, The common fragment of CTL and LTL, Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS'00), 2000, pp. 643–652.
[47] C. Wang, G.D. Hachtel, F. Somenzi, Abstraction Refinement for Large Scale Model Checking, Series on Integrated Circuits and Systems, Springer, 2006.
[48] C. Urban, A. Miné, Proving guarantee and recurrence temporal properties by abstract interpretation, Proceedings of the International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI'15), 2015, pp. 190–208.
[49] F. Wolter, M. Wooldridge, Temporal and dynamic logic, J. Indian Counc. Philos. Res. XXVII (1) (2011) 249–276.
[50] K.J. Turner, Abstraction and analysis of clinical guidance trees, Biomed. Inform. 42 (2) (2009) 237–250.
[51] J.C. Campos, J. Machado, E. Seabra, Property patterns for the formal verification of automated production systems, Proceedings of the 17th World Congress The International Federation of Automatic Control (IFAC'08), 2008, pp. 5107–5112.
[52] M.B. Dwyer, G.S. Avrunin, J.C. Corbett., Specification Patterns Website, 2018. Available at: < http://patterns.projects.cis.ksu.edu/ > (last visited on February 2019).
[53] D. Déharbe, Techniques for temporal logic model checking, First Pernambuco Summer School on Software Engineering (PSSE'04), Lecture Notes in Computer Science, vol. 3167, 2004, pp. 315–367.
[54] B. Pérez, I. Porres, Reasoning about UML/OCL models using constraint logic programming and MDA, Proceedings of the Eighth International Conference on Software Engineering Advances (ICSEA'13), 2013, pp. 228–233.
[55] B. Pérez, I. Porres, An overall framework for reasoning about UML/OCL models based on constraint logic programming and MDA, Int. J. Adv. Softw. 7 (1&2) (2014) 370–380.
[56] B. Pérez, I. Porres, Reasoning about UML/OCL class diagrams using constraint logic programming and formula, Inform. Syst. (2018) 1–26.
[57] B. Pérez, An MDE approach for reasoning about UML state machines based on constraint logic programming, Proceedings of the Ninth International Conference on Software Engineering Advances (ICSEA'14), 2014, pp. 34–39.
[58] E.K. Jackson, T. Levendovszky, D. Balasubramanian, Reasoning about metamodeling with formal specifications and automatic proofs, Proceedings of the 14th International Conference of Model Driven Engineering Languages and Systems (MODELS'11), 2011, pp. 653–667.
[59] E.K. Jackson, T. Levendovszky, D. Balasubramanian, Automatically reasoning about metamodeling, Softw. Syst. Model. (2013) 1–15.
[60] L. Fribourg, Constraint logic programming applied to model checking, Proceedings of the International Workshop on Logic Programming Synthesis and Transformation (LOPSTR'99), Lecture Notes in Computer Science, vol. 1817, Springer, 1999, pp. 30–41.
[61] E. Jackson, W. Schulte, Understanding specification languages through their model theory, Proceedings of the 17th Monterey conference on Large-Scale Complex IT Systems: development, operation and management, Lecture Notes in Business Information Processing, vol. 7539, 2012, pp. 396–415.
[62] L. Momtahan, A Simple Small Model Theorem for Alloy, Tech. Rep. RR-04-11, Oxford University Computing Laboratory, June 2004.
[63] K. Lausdahl, Enhancing Formal Modelling Tool Support with Increased Automation (Ph.D. thesis), Aarhus University, 2012.
[64] E.K. Jackson, N. Bjørner, W. Schulte, Canonical regular types, in: Proceedings of the 27th International Conference on Logic Programming, Technical Communication (ICLP'11), 2011, pp. 73–83.
[65] E.K. Jackson, E. Kang, M. Dahlweid, D. Seifert, T. Santen, Components, platforms and possibilities: towards generic automation for MDA, in: Proceedings of the 10th ACM international conference on Embedded software (EMSOFT'10), 2010, pp. 39–48.
[66] D. Harel, Statecharts: a visual formulation for complex systems, Sci. Comput. Program. 8 (3) (1987) 231–274.
[67] S. Ali, H. Hemmati, N.E. Holt, E. Arisholm, L. Briand, Model Transformations as a Strategy to Automate Model-Based Testing – A Tool and Industrial Case, Tech. Rep. 2010-01, Simula Research Laboratory, 2010.
[68] X. Devroey, M. Cordy, P.Y. Schobbens, A. Legay, P. Heymans, State machine flattening, a mapping study and tools assessment, in: Proceedings of the IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW'15), 2015, pp. 1–8.
[69] R.M. Gregorut, Synthesising Formal Properties from Statechart Test Cases (Ph.D. thesis), Institute of Mathematics and Statistics, University of Sao Paulo, 2015.
[70] J. Dubrovin, T. Junttila, Symbolic model checking of hierarchical UML state machines, in: Proceedings of the Eighth International Conference on Application of Concurrency to System Design, 2008, pp. 108–117.
[71] A. Wasowski, Flattening statecharts without explosions, Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04), vol. 39, ACM, 2004, pp. 257–266.
[72] K. Bogdanov, Automated testing of Harel's statecharts (Ph.D. thesis), Department of Computer Science, University of Sheffield, 2000.
[73] N.E. Holt, E. Arisholm, L. Briand, An Eclipse Plug-In for the Flattening of Concurrency and Hierarchy in UML State Machines, Tech. Rep. 2009-06, Simula Research Laboratory, 2009.
[74] E. Posse, PapyrusRT: modelling and code generation (invited presentation), in: Proceedings of the International Workshop on Open Source Software for Model

Driven Engineering (OSS4MDE'15), 2015, pp. 54–63.
[75] B. Selic, The pragmatics of model-driven development, IEEE Softw. 20 (5) (2003) 19–25.
[76] MOFScript User Guide, Version 0.9 (MOFScript v 1.4.0), 2006. Available at: < https://prototizer.com/userfiles/files/MOFScript-User-Guide-0_9.pdf > (last visited on February 2019).
[77] OMG, Mofscript Second Revised Submission to the MOF Model to Text Transformation RFP, OMG document ad/2005-11-03. Available at: < http://www.omg.org/ > (last visited on February 2019).
[78] The Eclipse UML2 project, website: < http://www.eclipse.org/modeling/mdt/?project=uml2 > (last visited on February 2019).
[79] Borland Together 2008 Edition for Eclipse, website: < https://supportline.microfocus.com/documentation/books/Together/2008R4/readme_together.html > (last visited on February 2019).
[80] Supplementary material. Specification of Requirements. Available at: < https://www.unirioja.es/cu/beperev/For-mulaSpecificationOfRequirements.html > (last visited on February 2019).
[81] Agency for Healthcare Research and Quality. National Guideline Clearinghouse. Available at: < http://www.guideline.gov > (last visited on February 2019).
[82] P. Groot, A. Hommersom, P. Lucas, R. Serban, A. ten Teije, F. van Harmelen, The role of model checking in critiquing based on clinical guidelines, Proceedings of the Eleventh European Conference on Artificial Intelligence in Medicine (AIME'07), Lecture Notes in Computer Science, vol. 4594, Springer, Amsterdam, The Netherlands, 2007, pp. 411–420.
[83] B. Bordbar, K. Anastasakis, UML2ALLOY: a tool for lightweight modelling of discrete event systems, Proceedings of the IADIS International Conference on Applied Computing (AC'05), 2005, pp. 209–216.
[84] K. Anastasakis, B. Bordbar, G. Georg, I. Ray, UML2Alloy: a challenging model transformation, Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07), Lecture Notes in Computer Science, vol. 4735, Springer, 2007, pp. 436–450.
[85] D. Jackson, Software Abstractions: Logic, language, and Analysis, MIT Press, 2006.
[86] M.B. Dwyer, G.S. Avrunin, J.C. Corbett, Property specification patterns for finite-state verification, Proceedings of the Second Workshop on Formal Methods in Software Practice (FMSP'98), ACM, 1998, pp. 7–15.