META-GLARE: A Meta-Engine for Executing Computer Interpretable Guidelines

Alessio Bottrighi[1(✉)], Stefania Rubrichi[1,2], and Paolo Terenziani[1]

[1] Computer Science Institute, DISIT, Univ. Piemonte Orientale, Alessandria, Italy
{alessio.bottrighi, stefania.rubrichi, paolo.terenziani}@uniupo.it
[2] Laboratory for Biomedical Informatics "Mario Stefanelli", Dipartimento di Ingegneria Industriale e dell'Informazione, University of Pavia, Pavia, Italy

Abstract. Clinical practice guidelines (CPGs) play an important role in medical practice, and computerized support to CPGs is now one of the most central areas of research in Artificial Intelligence in medicine. In recent years, many groups have developed different computer-assisted management systems of Computer Interpretable Guidelines (CIGs). We propose a generalization: META-GLARE is a "meta"-system (or, in other words, a shell) to define new CIG systems. It takes as input a representation formalism for CIGs, and automatically provides acquisition, consultation and execution engines for it. Our meta-approach has several advantages, such as generality and, above all, flexibility and extendibility. While the meta-engine for acquisition has been already described, in this paper we focus on the execution (meta-)engine.

Keywords: Computer interpretable guideline (CIG) · Metamodeling for healthcare systems · Meta CIG system · System architecture · CIG execution

1 Introduction

Clinical practice guidelines (CPGs) represent the current understanding of the best clinical practice. In recent years the importance and the use of CPGs are increasing in order to improve the quality and to reduce the cost of health care. ICT technology can further enhance the impact of CPGs. Thus, in the last twenty years, many different systems and projects have therefore been developed in order to manage computer interpretable CPGs. A survey and/or a comparative analysis of these systems are outside the goals of this paper. A comparison of Asbru, EON, GLIF, Guide, PROforma, PRODIGY can be found in [1]. In [2] the comparison has been extended to GLARE and GPROVE. The books [3, 4] represent a quite recent consensus of a part of the computer-oriented CIG community. A recent and comprehensive survey of the state-of-the-art about CIG has been published by Peleg [5].

The surveys/books show that few important commonalities have been reached. In particular, most approaches model guidelines in terms of a Task-Network Model (TNM): a (hierarchical) model of the guideline control flow as a network (graph) of specific tasks

D. Riaño et al. (Eds.): KR4HC/ProHealth 2015, LNAI 9485, pp. 37–50, 2015.
DOI: 10.1007/978-3-319-26585-8_3

(represented by nodes). Although the terminology may differ, all approaches support a basic set of core guideline tasks, such as decisions, actions and entry criteria. From the architecture point of view, most CIG approaches provide specific support for at least two subtasks: (i) CIG acquisition and representation and (ii) CIG execution, providing engines which support the execution of an acquired CIG on a specific patient. However, there are also important distinguishing features between the different CIG systems, often due to the fact that many of such systems are mostly research tools that evolve and expand to cover an increasing number of phenomena/tasks.

1.1 Origin and Motivation of the META-GLARE Approach

Such an evolution characterizes the history of GLARE (Guideline Acquisition, Representation, and Execution) [6, 7], the prototypical system we have been building since 1996 in cooperation with ASU San Giovanni Battista in Turin, one of the major hospitals in Italy. In our experience, it is quite frequent that, due to the need of facing a new real-world clinical guideline domain, some extensions to a CIG system are needed. In complex CIG systems (like GLARE) extensions require a quite large amount of work, since different parts of the code system must be modified, and their interactions considered. Specifically, extensions to the representation formalisms always involve the need of modifying the code of the acquisition, the consultation, and the execution engines of the system. On the other hand, the possibility of easily and quickly extend systems, and, more generally, of achieving ***fast prototyping*** when approaching new domains and\or tasks are essential in this field of AI in medicine research.

With such goals in mind, we started to re-design GLARE. Initially, we wanted to design yet a new CIG system, based on a new CIG formalism, enclosing the "best features" of current CIG approaches in the literature. However:

(1) such a general formalism would certainly be very general, and complex. However, CIGs have to be managed by physicians, so that simplicity (also in terms of the number of representation primitives being proposed) is a strict requirement. (2) the long-term experience of AI research has definitely shown that there is no "perfect" formalism. Whatever general CIG formalism could be defined, for sure can still require extensions, when facing new phenomena.

As a consequence of (1) and (2), we decided to pursue a completely different and innovative goal: instead of defining "yet another new CIG formalism and system", we chose to devise a "meta-system" (called META-GLARE), or, in other words, a shell supporting the definition of new CIG formalisms and systems (or facilitating the extensions of them). Though this idea is entirely new in the CIG literature, it stems from software engineering consolidated methodologies, and from the recent meta-modeling field in the medical informatics (see Sect. 6).

Our meta-system, called META-GLARE

(i) makes "minimal" assumptions as regards the CIG formalisms (basically, it ***simply assumes that CIGs are represented through TNM***)

(ii) ***provides general acquisition, consultation and execution engines, that are parametric over the specific CIG formalism being considered*** (in other words, the CIG formalism is an input of such engines).

1.2 Methodology and Advantages of the META-GLARE Approach

The core idea of our meta-approach is

(i) To define an open library of elementary components (e.g., textual attribute, Boolean condition, Score-based decision), each of which was equipped with methods for acquiring, consulting and executing it
(ii) To provide system-designers with an easy way of defining node and arc types (constituting the representation formalism of a new system) in terms of the elementary components constituting them
(iii) To devise general and basic tools for the acquisition, consultation and execution of CIGs, which are parametric with respect to the formalism used to represent them (in the sense that the definition of node and arc types are an input for such tools).

In such a way, we achieve several advantages:

- The definition of a new system (based on a new representation formalism) is easy and quick. Using META-GLARE, a system designer can easily define her/his own new system, by defining its formalism: (i) the node types, (ii) the arc types (both are defined types as an aggregation of components from the library), and (possibly) the constraints on them. No other effort (e.g., building acquisition or execution modules) is required.
- The ***extensions*** to an existing system (through the modification of its representation formalism) are easy and quick. In META-GLARE, a system designer can extend a system by defining and adding new node/arc types, or adding components to existing types (with no programming effort at all)
- ***User programming is needed only in case a new component has to be added in the component library***. However, the addition is ***modular*** and minimal: the system designer has just to focus on the added component, and to provide the code for acquiring, consulting, and (if needed) execute it. Such programming is completely "local" and "modular": the new methods have to be programmed "in isolation" (without having to care of the rest of the system). *No modification to any software component in META-GLARE architecture* (see Fig. 1 below) *is required to integrate the new methods*: META-GLARE automatically evokes them when needed during acquisition, consultation and execution.
- As a consequence, fast prototyping of the new (or extended) system is achieved (see the examples in Sect. 5).

We have already presented our innovative idea of proposing a "shell" for designing new CIG systems, and META-GLARE architecture, in the previous KR4HC workshop [8]. In such a paper, we have also described META-GLARE acquisition engine. This

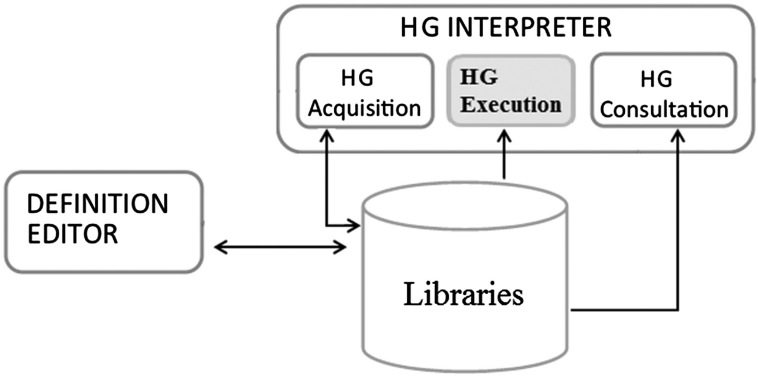

Fig. 1. The architecture of META-GLARE

paper is the natural completion of such a previous work, with the description of META-GLARE execution engine.

2 META-GLARE Architecture

META-GLARE supports any CIG representation formalism based on the following aspects: **(1)** guidelines are represented by **hierarchical graphs** (**constraints** on the graph –e.g., acyclicity- also supported); **(2)** there is no assumption on which **types** of **nodes** and **arcs** can be used to describe the graphs. The only assumption is that each **type** (of node and of arc) is defined as **a list of attributes**. **(3)** There is no assumption on which attribute types may be introduced in a specific formalism. We distinguish between two main categories of attributes: **control** attributes (i.e., those attributes that affect the execution of a node/arc; e.g., decision attributes) and **non-control** ones (e.g., textual attributes).

Thus, META-GLARE interpreter (guideline acquisition, consultation and execution tools) only assumes that a guideline is a hierarchical graph, and is ***parametrized*** on the **types of nodes** and **arcs,** and on the **types of attributes.** The interpretation of each node/arc type is obtained compositionally through the sequenced interpretation of the attributes composing it. This means that, practically, *each attribute type* (e.g., textual attribute, Boolean condition attribute, etc.) *must consists of the methods to acquire, consult, and (possibly) execute it*. Thus, for instance, guideline acquisition consists in the acquisition of a hierarchical graph, which in turn adopts the methods in each attribute type definition to acquire the specific attributes of the involved nodes/arcs.

In Fig. 1, we show a simplified version of the architecture of META-GLARE, focusing on parts affecting execution (for a more extensive description, see [8]). Oval nodes represent data structures, and rectangles represent computational modules. The DEFINITION_EDITOR tool supports system-designers in the definition of a new

system. It consists in four sub-components, to cope with the definition of (i) attribute types, (ii) node/arc types, (iii) graph constraints, and (iv) CIG formalism (where a CIG formalism is just a set of node/arc types and (possibly) of graph constraints). The output is an XML representation in the library.

Globally, the DEFINITION_EDITOR module manages the definition of the formalism a new CIG system. On the other hand, the HG_INTERPRETER (HG stands for "hierarchical graph") deals with the aspects which common to all the systems that can be generated by META-GLARE. It consists of three sub-components: HG_ACQUISITION, HG_CONSULTATION, and HG_EXECUTION. META-GLARE and its modules are developed as *Java Applets*. In this way, META-GLARE is a cross-platform application: it can be embedded into a web page and executed via web browsers without any installation phase. The libraries in Fig. 1 are implemented by databases stored in PostgreSQL. In the paper, we focus on the HG_EXECUTION module only.

3 CIG Execution Meta-Engine

The HG_execution module takes as input:

(1) A formalism (i.e., a set of arc/nodes types, each one consisting of a sequence of attributes)
(2) A specific CIG (the one to be executed), expressed in the given formalism
(3) A specific patient (whose data are collected in a database).

HG_execution supports the execution of the CIG on the specific data. Notice that, while all the execution engines in the CIG literature supports are specifically designed for the execution of a specific CIG formalism (so that their input are only (2) and (3) above), here the executor much more general, since any input formalism must be executable (i.e., also (1) is an input for the (meta-) executor). Our (meta-) executor only assumes that a guideline is a hierarchical graph, and is ***parametrized*** on the **types of nodes** and **arcs,** and on the **types of attributes.** As a consequence, it "inherits" from the CIG execution engines in the literature (see [9]) the way they deal with hierarchical graphs (points (i), (ii), and (iii.a) below), but it is parametric on the methods used to execute control attributes (point (iii.b) below). The basic idea in the definition of the (meta-) executor is simple:

(i) The execution of a CIG is the sequential execution of its nodes
(ii) Each node in the CIG is an instance of a type of node in the input formalism, and the node definition basically consists of a sequence of typed attributes
(iii) If the attribute is not a control attribute, (iii.a) it can be ignored by the executor. Otherwise, (iii.b) each (type of) control attribute has a method stating how it has to be executed. The executor simply execute such a method.

However, many refinements are required, concerning point (i) above (such refinements are considered also by most CIG engines in the literature). Basically, three

main problems have to be addressed: (1) the graph is hierarchical, (2) the graph is not simply a sequence of nodes (different types of arcs may be used; e.g., arcs for specifying alternatives, or concurrence), and (3) the flow of control may be altered by the execution of (the execution method of) a control attribute (e.g., the conditioned_GOTO or REPEAT attributes – see Sect. 3).

Regarding issue (1), since we assume that graphs may be hierarchical, we support the treatment of composite nodes, which are defined in terms of their components (which, in turn, are hierarchical graphs). The execution of a composite node starts with the execution of the first node of the sub-graph defining it, and ends when the execution of such a subgraph ends. Thus the nesting of calls to subgraphs must be explicitly managed by the executor.

As regards (2), point (i) naively assumes that graphs are defined only using one type of arc, representing the sequence in which two nodes have to be executed. However, notice that also the (types of) arcs are part of the definition of the input formalism. Thus, our engine must support the treatment of user-defined arcs, where each arc type is defined as a sequence of attributes. If an arc does not contain any control attribute, it can simply be ignored by the executor. On the other hand, if it has a control attribute, its method must be executed, to determine which is the next node to be executed. For the sake of brevity, in this paper we consider only directed arcs (as all CIG approaches do, to the best of our knowledge). We admit arcs with one starting node but multiple ending nodes (to support alternatives, concurrence, etc.). Thus, a set of nodes (to be executed) may be the result of the execution of an arc. Thus, also the fact that multiple actions may be candidate to be executed next must be managed by the executor.

As regards point (3), different types of control attributes can be used in a formalism. Our current library of attributes is briefly described in Sect. 3, and includes the main types provided by the CIG approaches in the literature [9]. However, we stress that such a library is *open*: new attribute types can be introduced when a new formalism (or an extension to a current formalism) is defined and no modification of HG_INTERPRETER is needed to manage them. Indeed, in many cases, such attributes can determine an "alteration" of the flow of control represented through the arcs in the hierarchical graph. For instance, an attribute type modeling repetition (REPEAT in our current library) can state that the next node to be executed is the current node itself; an attribute type modeling conditional GOTO (ConditionedGOTO in our current library) can state that the next node to be executed is the another node in the CIG, and so on. These alteration of the "standard" control flow of the graph must be managed, too, by the executor.

In order to cope with issues (1) and (2) above, the executor adopts a data structure (the ***execution_tree***) to explicitly represent, at each step, the hierarchy of active composite nodes (which are represented by the internal nodes of the tree) and the set of atomic nodes which are candidate to be executed next (the are represented by the leaves of the tree). The root of the tree is (by default) a "dummy" node representing the whole CIG, and each node in the tree is a "pointer" to a node in the CIG being executed.

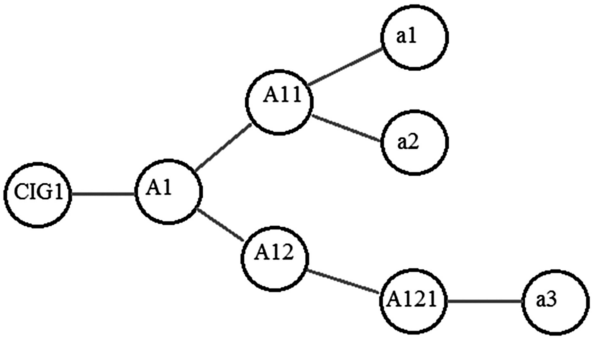

Fig. 2. Execution_tree: an example.

Of course, the execution_tree must be properly updated by the executor after the execution of each atomic node.

For example, the tree in Fig. 2 represents a situation in which CIG1 is being executed. The current composite action being executed is A1. The composite nodes A11 and A12 are the components of A1 currently being executed (concurrently). In turn, the currently executable (atomic) nodes in A11 are the atomic nodes a1 and a2. On the other hand, the active node composing A12 is still a composite node (A121), and its active component is the atomic node a3.

Concerning issue 3, though the library of attributes is open, we impose the constraint that each execution method in the library must return to the executor an indication of whether and how they affect the control flow. Six cases are considered (covering, to the best of our knowledge, the main possibilities considered by CIG execution engines in the literature [9]; see the discussion in Sect. 6):

(1) "**go_on**": this is the standard continuation. With a "go_on", the executor must execute the next control attribute in the node (or has finished the execution of the node, if the current attribute is the last control one) (Algo 1 line 19)
(2) "**repeat**": the current control attribute must be executed again (Algo 1 lines 9–11)
(3) "**suspend**": the execution of the current CIG has to be suspended (Algo 1 lines 12–13)
(4) "**abort**": the execution of the current CIG has to be stopped, and terminates. (Algo1 line 16)
(5) "**goto**" <node>: the execution of the CIG restarts from the execution of <node> (Algo 1 line 18)
(6) "**fail**": a failure has occurred (e.g., an action could't be executed, because the required instrument is not available). The current execution is stopped, and the executor must enact the general recovering facility. (Algo 1 line 17).

The execution of a CIG starts with the initialization of the execution tree. Then, the executor operates as abstractly described by the algorithm Algo 1.

```
1.  Executor_algo(nodes/arc types definitions, library of attribute types, CIG, patient
    data, Et: execution tree)
2.  while (not null Et) do
3.  begin
4.          Cur_n ← choose one leaf from Et;
5.          Modality ← "go_on";
6.          while ((there are control attributes in Cur_n) AND (modality = "go_on")) do
7.          begin
8.                  Modality←exec(control_attribute.execmethod)
9.                  if      (Modality = "repeat") then
10.                         repeat Modality← exec(control_attribute.execmethod)
11.                         until Modality≠"repeat";
12.                 if      (Modality = "suspend") then
13.                         wait until modality = "go_on"
14.         end;
15.         case Modality of
16.                 "abort": delete(Et) Exit;
17.                  "fail": manage failure;
18.                  "goto" <node>: Et←construct_new_tree(<node>,CIG);
19.                 "go_on": update_tree(Cur_n, Et, CIG);
20.         endcase;
21. end.
```

Algorithm 1. Pseudo-code of the algorithm for the CIG execution

Leaves of the execution_tree represent possibly concurrent actions, so that it is up to the user-physician to select which leaf has to be executed next (Algo 1 line 4). In the case of "goto" (Algo 1 line 18), the current execution tree is substituted by a new tree, in which a "pointer" to <node> is a leaf. However, also the "upper parts" of such a tree must be built (considering the composite actions in the CIG containing <node>, if any), by *construct_new_tree*. The "standard" continuation ("go_on" modality, Algo 1 line 19) is managed through a proper update of the execution tree, as shown by Algo 2 in the Appendix.

4 Control Attributes

In META-GLARE, two basic categories of control attribute types can be defined: the control attributes of nodes, and the one of arcs.

4.1 Control Attributes of Nodes

In our approach, attribute types are characterized by several *features*, which are specified according to the XML document. A XML tag, which describes an *attribute type*, has several *features*, defining its name, its properties, and its interpretation. "Interpretation" tags are very important, since they define (pointers to) the methods that are used by the HG_INTERPRETER to acquire, store, consult and execute any instance of such types.

Notably, the XML definition of "procedural" attributes does not contain the Java code of the methods, but only symbolic pointers to them.

We have currently several control attributes types. However, the library is open, and new attributes can be added by users. Indeed, the only constraint is that the execution methods must return as output one of the five modalities discussed in Sect. 3. At the current stage, we have implemented: BooleanCondition, BooleanDecision, ScoredCondition, ScoredDecision, DataEnquiry, FAIL, ConditionedFAIL, ABORT, ConditionedABORT, GOTO, ConditionedGOTO, REPEAT, external_action. Names are mostly self-explicative. Conditions are evaluated on the basis of patient's data automatically retrieved from a database. Decisions (between alternatives) are evaluated by evaluating the condition associated with each one of the alternatives, and showing to the user the result of the automatic evaluation. The final decision between alternatives is left to the physicians. DataEnquity is a data request. Data are retrieved from the database. If they are not present, the execution engine waits for them. External_action is a special control attribute type, to model those activities that have to be performed (e.g., by physicians, nurses, etc.) on the patient (e.g., the administration of a drug). The corresponding methods simply ask to the user whether the action has been successfully performed (returning the "go_on" modality), or not (returning the "fail" modality).

4.2 Control Attributes of Arcs

As discussed in Sect. 3, we support arcs having one starting node, and one or many ending nodes. Each arc is described by a set of attributes. Control arcs have exactly one control attribute (but they may have other non-control ones). The library is open, and currently contains the definition of three control attributes for arcs: ***sequence***, ***alternative***, and ***concurrence***. These three types cover the most significant cases considered in the literature, except "constrained" arcs in GLARE, which support complex temporal constraints between nodes. Sequence has just one ending node, and its execution method simply returns it. Concurrence has two or more ending nodes, and returns them. Alternative has two or more ending nodes, and asks to the user-physician to choose one of them (which is returned).

5 META-GLARE in Action

META-GLARE takes as input a CIG formalism (based on the TNM model) and provides as output a CIG system to acquire and execute guidelines represented on top of such a formalism. Thus, it support fast CIG system prototyping, both when building a new CIG system (on the basis of a new TNM-based formalism), and when extending a current one (with the addition of new features to a CIG formalism).

Example 1. As a first concrete example, suppose that, while analysing a new domain, a system designer identifies the need of enriching her\his CIG formalism (or to design a new CIG formalism) with a new type of node, consisting of

- some non-control attributes (e.g., textual attributes for name and goal, numerical attribute for cost, and so on)
- a sequence of three control attributes, "precondition" (of type ConditionalFAIL), "body" (of type external_action) and "postcondition" (of type ConditionalFAIL).

In particular, the intended purpose of (the control attributes of) the node is that, during execution, preconditions (which are Boolean conditions) are first checked on the patient data. If they are not satisfied, the execution of the node fails (and fail recovery is started); otherwise, an external action is executed on the patient. After the execution, post-conditions are verified, possibly leading to a failure of the execution of the node.

All the required attribute types in the example are already present in META-GLARE Library (textual and numerical types for non-control attributes; type ConditionFAIL for "preconditions" and "postconditions"; type "external_action" for the external action). As a consequence, the system-designer has simply to enter the new node definition (through the graphical interface of the DEFINITION_EDITOR). No other effort is required. The whole work requires only few minutes to the system-designer. As a result, the executor described in Sect. 3 will deal also with CIGs having such a new type of nodes, without requiring any modification.

On the other hand, some programming is required in case a new attribute type (not already existent in the library) has to be added.

Example 2. When choosing among clinically "equivalent" (with respect to the patient at hand) therapies, the "long term" effects of the therapeutic choice (e.g., what path of actions should be performed next, what are their costs, durations and expected utilities) may be helpful to discriminate. Decision theory [10] may be helpful in this context. In particular, it allows to identify the optimal policy, i.e., the (sequence of) action(s) changing the states of the system in such a manner that a given criterion is optimized. In order to compare the different action outcomes, one commonly assigns a utility to each of the reached states. Since there is uncertainty in what the states will be, the optimal policy maximizes the expected utility. It has been recently shown how decision theory can be exploited in the CIG context, to model *cost/benefit decisions* [11]. Currently, no facility for cost/benefit decisions is provided in META-GLARE Library. Thus, to extend a (META-GLARE-based) CIG formalism with a new type of control attribute (say Cost/Benefit_Decision) modelling cost/benefit decisions, new methods must be developed by the system designer, to acquire, consult, and execute it. But ***no*** modification to META-GLARE acquisition, consultation and execution engines has to be performed. In particular, focusing on execution, it is fundamental to stress that, as long as the new execution method Cost/Benefit_Decision returns one of the six modalities managed by the executor, no modification to the executor itself is needed at all.

6 Related Works, Conclusions, and Future Work

In this paper, we propose a (partial) description of the execution engine of META-GLARE, an innovative approach to cope with CIGs.

The HG_INTERPRETER, discussed in Sect. 3 is general, in that it covers a family of different formalisms (all the formalism that can be defined by META-GLARE). In such a way, it is neatly different from all the others CIG interpreters in the literature, which are biased to the treatment of a specific formalism [9]. In particular, different CIG interpreters have been compared in [9], where it is highlighted that, with the only exception of PROforma [12], that uses Prolog interpreter for execution, all the other systems have developed their own formalism-dependent interpreter. However, some of them share common features. Systems such as Arezzo^{TM} [13], GLARE [7], HeCaSe2 [14], SAGE [15] and GLEE [16] use similar basic elements (actions, decisions and enquiries). In particular, to enhance the generality of our approach, the META-GLARE interpreter covers (to the best of our knowledge) all the main modalities considered by the different interpreters in the literature. Notably, our treatment of modality partly encompasses the state transition model for the execution of a single action of approaches like PROforma [12] and Asbru [17] (which, for instance, consider states like "abort" or "suspend"; other states, like "in progress" or "completed" represent the "standard" execution of an action, so that they are implicitly managed by our meta-interpreter).

The main idea underlying META-GLARE is simple: instead of proposing "yet another system" to acquire, represent and execute CIGs, we propose a "meta-system", i.e., a shell to define (or modify) CIG systems. Roughly speaking, the input of META-GLARE is a description of a representation formalism for CIGs, and the output is a new system able to acquire, represent, consult and execute CIGs described using such a formalism. Indeed, such a basic idea is not at all new in Computer Science, although it is the first time that it has been applied to the domain of CIGs.

A similar idea, in a completely different context, has emerged in Computer Science in the 70', with the definition of the so-called "compilers of compilers", like YACC (Yet Another Compiler of Compilers [18]). In particular, META-GLARE takes as input any CIG formalism and provides as output a CIG system (i.e., an acquisition, a consultation and an execution engine) for such formalism just as YACC takes as input any context free language (expressed through a formal attribute grammar) and provides as output a compiler for it. More recently, Model-Driven Software Engineering (MDSE) has emerged as a promising methodology for software systems, targeting challenges in software engineering relating to productivity, flexibility and reliability. MDSE is especially useful as a methodology for the development of healthcare systems, and even a dedicated workshop (the International Workshop on Metamodelling for Healthcare Systems, 2014 (http://mmhs.hib.no/2014/) and 2015 (http://mmhs.hib.no/2015/) has been created to face such a topic).

Indeed, the application of models to software development is a long-standing tradition, and has become even more popular since the development of the Unified Modeling Language (UML). Yet we are faced with 'mere' documentation, MDSE has an entirely different approach: Models do not constitute documentation, but are considered equal to code, as their implementation is (semi)automated. MDSE therefore aims to find domain-specific abstractions and makes them accessible through formal modeling. This procedure creates a great potential for automation of software production, which in turn leads to increased productivity, increasing both the quality and maintainability of software systems. We share part of the methodology of MDSE, such as the use of three levels of models (the meta-formalism level, the formalism level, and

the CIG instance level), but a relevant difference should be pointed out. In "standard" MDSE approaches, the final model is used to semi-automatically generate the application code, through the adoption of transformation rules. On the other hand, there is no semi-automatic code generation in META-GLARE. Indeed, the HG-interpreter is already provided by META-GLARE, but it is parametrized over the CIG formalism, so that a CIG interpreter is automatically obtained when a specific CIG formalism is selected.

To the best of our knowledge, the application of such ideas to the context of CIG is completely new. Such an application has mainly motivated by our goal of designing and implementing a flexible and powerful vehicle for research about CIG. In our opinion, META-GLARE provides two main types of advantages, both strictly related to the notion of *easy* and *fast prototyping*. Using META-GLARE

(1) the definition of a new system (based on a new representation formalism) is easy and quick;
(2) The extension of an existing system (through the modification of the representation formalism) is easy and quick.

In particular, the executor described in Sect. 3 will deal also with CIGs having such a new type of nodes, without requiring any modification. On the other hand, some programming is required in case a new attribute type (not already existent in the library) has to be added (see discussion in Sect. 5).

Thus, META-GLARE is, above all, a good vehicle for fast definition/extension and prototyping CIG systems, making it quite suitable especially as a research tool to address new CIG phenomena.

The implementation of META-GLARE execution engine is actually ongoing. We plan to finish it as soon as possible, to start with an extensive experimental evaluation of our approach. Though quite powerful, the current approach has several limitations, which we want to overcome in our future work. In particular, we want to consider the addition of new modalities (besides the ones discussed in Sect. 3), and we aim to extend our current approach to deal (i) with exceptions (along the lines discussed in [19] and (ii) with the concurrent (but possibly interacting) execution of two or more CIGs, to cope with comorbidities (integrating the work in [20] into META-GLARE)).

Acknowledgements. The research described in this paper has been partially supported by Compagnia San Paolo, within the GINSENG project.

Appendix

Algorithm Algo 2 in the following describes how to update of the execution tree, in case of "go_on" modality.

```
1.  update_tree(Node,Tree,CIG)
2.  begin
3.          Mother_n←get_mother(Node,Tree);
4.          delete(Node,Tree);
5.          if Node has brothers then return Tree;
6.          else
7.                  begin
8.                  Next_nodes←get_next(Node,CIG);
9.                  if (Next_nodes ≠ null) then
10.                         for each Node∈ Next_nodes
11.                         do append(expand_down(Node,CIG), Mother_n);
12.                         return Tree;
13.                 else return update_tree(Mother_n, Tree,CIG)
14.                 end
15. end.
```

Algorithm 2. Pseudo-code of the algorithm for the update of the execution tree, in case of "go_on" modality.

Once a node has been executed, it is deleted from the execution tree. In case there are concurrent nodes to be executed (brothers) (line 5), the executor simply has to operate on such a new tree. Otherwise (lines 7–13), the deleted node has to be substituted by the immediately-next nodes to be executed in the CIG (in the case of concurrent actions, there is more then one "immediatly-next" node to be considered). The function ***get_next*** consider the control arc (which must be unique, if it exist) exiting from Node in the CIG, and execute it is execution method (line 8). As a result, a set of next nodes to be executed is returned. Each one of such nodes must be added to the tree (***append*** function), and possibly expanded (***expand_down***: if Node is composite, then the first nodes (in the case of concurrent actions, there is more then one "first" node to be considered) of the CIG subgraph representing it are appended to treeNode, and so, on, recursively, until atomic nodes are reached, lines 10–12). On the other hand (line 13), if there are no next node (i.e., if the executed node was the last one in a graph or subgraph), then the update_tree algorithm must be recursively applied on the mother of the current node.

References

1. Peleg, M., Tu, S., Bury, J., Ciccarese, P., Fox, J., Greenes, R.A., Hall, R., Johnson, P.D., Jones, N., Kumar, A., Miksch, S., Quaglini, S., Seyfang, A., Shortliffe, E.H., Stefanelli, M.: Comparing computer-interpretable guideline models: a case-study approach. JAMIA **10**(1), 52–68 (2003)
2. Bottrighi, A., Chesani, F., Mello, P., Montali, M., Montani, S., Storari, S., Terenziani, P.: Analysis of the GLARE and GPROVE approaches to clinical guidelines. In: Riaño, D., ten Teije, A., Miksch, S., Peleg, M. (eds.) KR4HC 2009. LNCS, vol. 5943, pp. 76–87. Springer, Heidelberg (2010)

3. ten Teije, A., Miksch, S., Lucas, P. (eds.): Computer-Based Medical Guidelines and Protocols: a Primer and Current Trends. IOS Press, Amsterdam (2008)
4. Lucas, P., Hommerson, A. (eds.): Foundations of Biomedical Knowledge Representation. Springer, Heidelberg (2015)
5. Peleg, M.: Computer-interpretable clinical guidelines: a methodological review. J. Biomed. Inform. **46**(4), 744–763 (2013)
6. Terenziani, P., Molino, G., Torchio, M.: A modular approach for representing and executing clinical guidelines. Artif. Intell. Med. **23**(3), 249–276 (2001)
7. Terenziani, P., Montani, S., Bottrighi, A., Molino, G., Torchio, M.: Applying artificial intelligence to clinical guidelines: the GLARE approach. In: [3], 273–282 (2008)
8. Terenziani, P., Bottrighi, A., Lovotti, I., Rubrichi, S.: META-GLARE: a meta-system for defining your own CIG system: architecture and acquisition. In: Miksch, S., Riano, D., ten Teije, A. (eds.) KR4HC 2014. LNCS, vol. 8903, pp. 95–110. Springer, Heidelberg (2014)
9. Isern, D., Moreno, A.: Computer-based execution of clinical guidelines: A review. Int. J. Med. Inform. **77**, 787–808 (2008)
10. Russel, S., Norving, P.: Artificial Intelligence: a Modern Approach. Prentice Hall, New Jersey (2009)
11. Anselma, L., Bottrighi, A., Molino, G., Montani, S., Terenziani, P., Torchio, M.: Supporting knowledge-based decision making in the medical context: the GLARE approach. IJKBO **1** (1), 42–60 (2011)
12. Sutton, D.R., Fox, J.: The syntax and semantics of the PROforma guideline modeling language. J. Am. Med. Inform. Assoc. **10**, 433–443 (2003)
13. InferMed, Arezzo Technical White Paper, Technical report InferMed, Ltd. http://www.infermed.com/ Accessed 18 May 2015
14. Isern, D., Sánchez, D., Moreno, A.: HeCaSe2: A Multi-agent Ontology-Driven Guideline Enactment Engine. In: Burkhard, H.-D., Lindemann, G., Verbrugge, R., Varga, L.Z. (eds.) CEEMAS 2007. LNCS (LNAI), vol. 4696, pp. 322–324. Springer, Heidelberg (2007)
15. Tu, S.W., Campbell, J.R., Glasgow, J., Nyman, M.A., McClure, R., McClay, J., Parker, C., Hrabak, K.M., Berg, D., Weida, T., Mansfield, J.G., Musen, M.A., Abarbanel, R.M.: The SAGE guideline model: achievements and overview. JAMIA **14**(5), 589–598 (2007)
16. Wang, D., Peleg, M., Tu, S.W., Boxwala, A.A., Ogunyemi, O., Zeng, Q., Greenes, R.A., Patel, V.L., Shortliffe, E.H.: Design and implementation of the GLIF3 guideline execution engine. J. Biomed. Inform. **37**, 305–318 (2004)
17. Young, O., Shahar, Y., Liel, Y., Lunenfeld, E., Bar, G., Shalom, E., Martins, S.B., Vaszar, L.T., Marom, T., Goldstein, M.K.: Runtime application of Hybrid-Asbru clinical guidelines. J. Biomed. Inform. **40**, 507–526 (2007)
18. Johnson, S.C.: Yacc: Yet Another Compiler-Compiler, vol. 32. Bell Laboratories, Murray Hill, NJ (1975)
19. Leonardi, G., Bottrighi, A., Galliani, G., Terenziani, P., Messina, A., Della Corte, F.: Exceptions handling within GLARE clinical guideline framework. AMIA Annu. Symp. Proc. **2012**, 512–521 (2012)
20. Piovesan, L., Molino, G., Terenziani, P.: Supporting Physicians in the Detection of the Interactions between Treatments of Co-Morbid Patients, In: Tavana, M., Ghapanchi, A.H., Talaei-Khoei A. (Eds.) Healthcare Informatics and Analytics: Emerging Issues and Trends, Chapter: 9, IGI Global (2014)