A goal-oriented framework for specifying clinical guidelines and handling medical errors

Adela Grando¹, Mor Peleg²³, David Glasspool¹

¹ School of Informatics, University of Edinburgh, Edinburgh, UK
² Stanford Center for Biomedical Informatics Research, Stanford University, Stanford, USA
³ Department of Management Information Systems, University of Haifa, Haifa, Israel

Address Correspondence to:
Adela Grando
University of Edinburgh
School of Informatics
10 Crichton Street, Informatic Forum
EH8 9AB, Edinburgh (UK)
Fax: (131) 6506899
mgrando@staffmail.ed.ac.uk
ABSTRACT

Computer-interpretable guidelines (CIGs) aim to improve patient care and reduce medical errors. Although CIGs implement evidence-based recommendations they cannot prevent exceptional behavior from happening. To address this problem we developed a framework that can monitor, detect, and handle exceptions that occur during normal CIG execution and can potentially prevent them from developing into medical errors. Our framework enables specifying the goals of a guideline and linking them with recommended tasks that could satisfy the goals. Exceptions are linked with goals that manage them, which can be realized by tasks or plans. To achieve a link between the tasks, plans, goals, monitored effects, and exceptions, our definition of goals and exceptions is state-based. We demonstrate our approach using a generic plan for management of a chronic disease and a particular instantiation for hypertension management.

Keywords: clinical guidelines, exceptions, exception handling, goals, medical errors
I. INTRODUCTION

Clinical guidelines are systematically developed statements to assist practitioner and patient decision making about appropriate care for specific clinical circumstances [1]. They aim to provide evidence-based recommendations in order to limit medical errors, limit unjustified practice variation, and reduce costs. Clinical guidelines can potentially contribute to the implementation of more effective, safer, and more efficient evidence-based clinical care, allowing errors, inconsistencies, incompleteness, or inefficiencies to be detected and decision-support to be provided, and have been suggested by the Institute of Medicine (IOM) [2] (p. 145) as one of the means for promoting patient safety through standards of practice.

Studies have shown that guideline implementations can best affect clinician behavior if they deliver patient-specific advice during the clinical encounter [3, 4]. Computer-interpretable guideline (CIG) specification languages have been developed for this purpose [5].

Despite much research and many successful trials of CIGs, few such systems are routinely used in clinical practice. There are many barriers to CIG implementation and adoption including, among others, cost of implementation, interaction with electronic medical records (EMR), computer literacy and hardware/software requirements, and impact on clinician workflow [6]. Many of these potential barriers may be overcome with appropriate system design [7]. However, important barriers that have not yet been addressed include the ability to provide decision-support in the case of failures and exceptions as well as when the encoded guideline knowledge is incomplete. One major concern for CIG research is how to deal with the large deviations from the expected course of events that can arise from medical errors and other unanticipated events and their handling. Some work has been done towards increasing the flexibility of clinical workflow definitions by specifying generic workflows that can be specialized at run time, based
on alternative methods for achieving a clinical goal [8, 9] or for responding to a scenario [10]. In this paper we present a generic approach to handling exceptions in workflow execution (that is, any flow of execution or result of execution in a workflow that deviates from the expected nominal course of events). Our approach is based on a flexible workflow definition schema using clinical goals at multiple hierarchical levels [9], and separates exception detection and handling cleanly from normal workflow execution. Our approach may be applied even when the represented guideline knowledge is incomplete.

In common with Greenberg et al.[11] we argue that it is unrealistic to expect to prevent all predictable errors or adverse events, but that it is realistic to aim instead to detect all significant deviations from the expected nominal execution of the guideline at run time so that corresponding remedial actions may be taken. Our framework therefore defines an exception handler (comprising a hierarchical catalog of state-based exceptions) at design time and a monitoring system detects exceptions using this catalog at run-time. The design of the catalog is independent of the workflow specification; the exception-manager system does not associate states corresponding to exceptions with points in the workflow until run-time. In this way, normal flows are separated from exceptional flows to allow more modular, clear, and easy to maintain workflow specifications. Each exception in the catalog is associated with a goal to manage the medical error or other situation described by the exception.

A novelty of our work is that we classify exceptions into obstacles and hazards, in the spirit of the definition of medical errors presented by the Agency for Healthcare Research and Quality in the US Department of Health and Human Systems in [12]. According to this definition, medical errors are "mistakes made in the process of care that result in or have the potential to result in harm to patients; including the failure of a planned action to complete as intended or the use of a
wrong plan to achieve an aim..." [12]. While the use of safety rules to activate prophylactic treatments and warnings [13, 14] may allow avoidance of some harmful actions at run-time, we introduce the notion of a hazard to detect any possibly harmful states that nonetheless do occur at run-time so that compensatory actions may be executed if the hazard is confirmed to be a real threat to patient safety. We say that there is an obstacle on the other hand when the guideline could not be followed, or is followed but does not have the intended effect (for instance, it was not possible to complete the enactment of a task).

We begin in section II by setting out a goal-based clinical workflow framework within which to frame our proposal, and outlining our approach to handling exceptions. We illustrate the framework by developing an example of generic workflow for chronic diseases in section III.

In section IV we show how an exception manager may be defined for the example chronic disease workflow, giving examples of obstacle and hazard definitions. Finally we discuss the application of the system, related research and future work.

II. DESCRIPTION OF THE FRAMEWORK

We use a goal-based approach to deal with both normal and exceptional clinical workflows. Goal-based planning of CIGs is not a new approach [8, 9, 15]. The goal-based planning approach underlying our framework [9] borrows some basic notions from PROforma [16,17], a clinical workflow language that is not itself goal-based. PROforma bases its process model on a minimal ontology of task classes that can be composed into networks representing arbitrarily complex plans. There are four main task classes derived from the root class in the task ontology (called keystone). Actions represent procedures to be executed on the external environment while enquiries are specialized actions carried out to acquire information from some person or external
system. Decisions define choices about what to believe or what to do. Plans group together a set of tasks (including other plans) and define their scheduling.

PROforma is an executable language with an execution semantics based on a simple state transition system. The root keystone task type has an attribute state which can take a small set of values at run time to indicate the state of the task: Dormant, InProgress, Completed or Discarded.

The following description of our framework is based around these notions but we will highlight the main areas in which we deviate from standard PROforma.

A. Workflows

In our framework a workflow is a network of keystones, connected by scheduling constraints based on Petri Nets (sequential composition, XOR-split, AND-split, XOR-join, and AND-join). Workflows have a unique starting point and ending point. The notion of keystone as the root task type is borrowed from PROforma; we add more expressive Petri Net based scheduling constraints.

1. Keystones

Keystone is the root class of the task hierarchy. Formally a keystone is defined as the tuple

<name, parameters, antecedents, state, precondition, incomingTriggers, startAt, duration, finishAt, cycleInterval, parentPlan, roles, actor, successCondition, abortCondition, outgoingTrigger>

Most of these attributes are inherited from the keystone class in PROforma with the same or slightly modified semantics. We add five attributes: duration, finishAt, roles, actor, outgoingTriggers. In our framework we interpret the keystone attributes as follows:

- Name: a unique natural language label.
• **Parameters**: formal input parameters.

• **Antecedents**: list of incoming scheduling constraints.

• **State**: a keystone can be in state Dormant, InProgress, Discarded, Completed or Suspended, as shown in the state transition system of Figure 1. Initially a keystone is Dormant and it changes to state InProgress when it is enacted. While the **successCondition** of the keystone is not satisfied it remains InProgress. The keystone changes from InProgress to Completed when the **successCondition** is satisfied. If a task is Dormant or InProgress and the **abortCondition** is satisfied it changes to the Discarded state. Compared with PROforma we reinterpreted the InProgress and Completed states and we added the new state Suspended. While in PROforma iterative tasks became Completed after each iteration is enacted, in our framework tasks remain InProgress, only becoming Completed when the last iteration is enacted and the **successCondition** is satisfied. While a Suspended keystone can be resumed from the execution point where it was interrupted, Discarded keystones can only be reinitialized (become InProgress) from the beginning, and only if the **stardCond** is satisfied. The Suspended state is introduced as part of the exception handling scheme and we explain its semantics in section II.B.

• **Precondition**: a predicate that must be satisfied before the keystone becomes InProgress.

• **IncomingTriggers**: a set of trigger messages that can be received by the keystone. If this set is not empty then at least one trigger must be received before the keystone can become InProgress.

• **startAt, duration, finishAt**: predicates that once evaluated return a time constraint, for
instance a date, a time interval, etc. They indicate the starting time, duration and completion time for the keystone.

- **CycleInterval**: indicates whether the keystone is cyclic and specifies its frequency of iteration.
- **ParentPlan**: the name of the plan, if any, which this keystone forms a part of.
- **Roles**: lists classes of actors who are allowed to take responsibility for executing the keystone.
- **Actor**: the actual actor (person or automatic system) responsible for performing the keystone, which must belong to the classes defined in roles. For example, if radiologist is in the set of roles for performing the keystone, then John who is a radiologist can enact the keystone.
- **SuccessCondition**: a predicate which, when satisfied, changes the state of the keystone from InProgress to Completed.
- **AbortCondition**: a predicate which, when satisfied, changes the state of the keystone from InProgress to Discarded. While in PROforma only plans could be aborted, in our framework any keystone may be aborted.
- **OutgoingTriggers**: The set of trigger messages that are fired (generated) by the keystone when it is in state InProgress.

We define execution semantics via the state attribute of the keystone class. Figure 1 shows the state transition system for a keystone. The four conditions startCond, successCond, cycleCond and abortCond on transitions in Figure 1 determine when a task can be started (or re-started in the case of a cyclic task or a goal which has not been achieved as expected by a candidate plan), can successfully complete, can start a new iteration, or should be aborted, respectively. These
conditions are based on the corresponding conditions for PROforma and are defined as follows:

StartCond is true iff:

1) the keystone is Dormant and its parent plan (if any) is InProgress, the keystone’s scheduling constraints are satisfied, the attributes precondition and StartAt are satisfied, and if the keystone has any incoming triggers defined at least one has been received; or

2) the keystone is a goal, a candidatePlan that should have achieved the goal (but did not) has been Completed, the goal’s successCondition is not satisfied, the goal is not cyclic, it is InProgress and its attributes precondition, startAt are satisfied and the time restrictions duration and finishAt are not violated;

abortCond is true iff any one of the following is true:

1) the keystone is Dormant and its abortCondition is true, or

2) the keystone is Dormant, its parentPlan is InProgress, its attributes precondition and startAt are true but its scheduling constraints are not satisfied because some antecedent keystone has been Discarded, or

3) the keystone is InProgress, its abortCondition is satisfied and one of the following conditions is true: the keystone is a task under enactment, or the keystone is a goal and none of its candidate plans is InProgress.

SuccessCond: is true iff the keystone is InProgress, its successCondition is satisfied and one of the following conditions is true: the keystone is a task under enactment, or the keystone is a goal and none of its candidate plans is InProgress.

CycleCond: is true if the keystone is cyclic, it is InProgress and the attributes cycleInterval, duration, finishAt are satisfied.

The final condition shown in Figure 1, SuspendingException, will be explained in detail in
section II.B.

2. Tasks

Tasks are, as in PROforma, a type of keystone that can change the workflow state including the flow of control in the workflow as well as the state of the environment (the scenario in [10]) when enacted. Tasks can be of four types defined as in PROforma: decision, enquiry, action, or plan (workflows comprising activities and goals).

Tasks inherit all the attributes of keystone and have three extra attributes:

- **goalsAchievable:** while in PROforma it is possible to indicate a “goal” condition that is expected to be true on task completion, this has very limited use within the language. In our framework we can formally specify a set of achievable goals which determine guideline enactment as discussed below.
- **enactmentType:** describes whether the task is manual or automatic.
- **Procedure:** if enactment type is automatic, this attribute should contain a reference to the procedure that automates the action. This attribute is not present in PROforma.

Additionally some types of tasks have their own set of task-specific attributes which are identical to those for PROforma (for example to specify decision parameters, data sources etc.).

3. Goals

Compared with PROforma a major addition in our framework is the goal task type. Goals are keystones that represent temporal patterns of state variables that need to be achieved, maintained, prevented or ceased. Formally, a goal inherits all the attributes of keystone and has the following further attributes:

- **GoalType:** specifies whether the goal is to achieve, maintain, prevent or cease a state. A state can be achieved if it is not satisfied but could be satisfied in the future. The satisfaction...
of a state may be maintained for some period of time or indefinitely. A state that is not satisfied may be prevented by maintaining its unsatisfied state for some period of time or indefinitely. A state that was satisfied is considered to have ceased when it is no longer satisfied. To allow more intuitive specification of goals, we find that we are able to express the goal type prevent using negation of the type maintain and the goal type cease using negation of the type achieve.

- **CandidatePlans**: corresponds to a list of plans that are recommended at run-time as candidates to achieve an InProgress goal. When the goal is InProgress and its successCondition and abortCondition are not satisfied, the decision-support system proposes candidate plans for satisfying the goal. The candidates are those plans that are retrieved during run time from a repository of plans and can change their state from Dormant to InProgress because Precondition, incomingTriggers and startAt restrictions are satisfied, as explained in section II.A. Once the candidate plan chosen for achieving a goal has been Completed the goal is still InProgress and its successCondition is checked.

- **InvariantCondition**: corresponds to a state that needs to be satisfied when the goal changes its state from Dormant to InProgress (and should be satisfied again when a plan from candidatePlans has been chosen and Completed). We have chosen this notion of invariant, instead of a condition that should be permanently true, in order to associate invariants with cyclic goals (see for example the goal maintain_chronic_disease_managed that we present in section III).

Having described our workflow framework, which is based in part on PROforma, we now move on to the exception system which is entirely new.
B. The exception monitoring and handling system

Abnormal or undesirable states can occur in an unpredictable way during the enactment of a clinical protocol. It can even be the case that undesirable states are not associated with a specific point in the workflow but instead emerge from complex workflow interactions. To detect and recover from these situations exceptions can be specified independently of the workflow, to be dynamically triggered at run time.

Otherwise, for predictable abnormal or undesirable situations, which are not necessarily frequent, subflows can be specified to enable detection of predictable undesirable situations and recovery from them. For instance, if there is evidence that in 10% of the cases a treatment could have undesirable side effects for the patient, it is reasonable to consider the detection and recovery of this situation as part of the workflow specification, and not as an exception.

We define two types of exceptions that can occur during guideline execution as follows:

A hazard corresponds to an unpredictable state that can potentially produce harm to the patient. For instance, a hazard might be detected when the user decides to deviate from the recommended plans to achieve a goal, or when the patient has developed an unexpected side effect that needs to be monitored.

An obstacle corresponds to a state where nominal execution of the guideline is unexpectedly not possible. For example the user prescribed the recommended treatment but the patient did not respond as expected, or the user started to perform a recommended task at a certain time but could not finish it because the resources required were not available.

When it is certain that an unpredictable workflow state will cause harm (e.g. when a drug combination has a negative effect on the patient’s condition, even when according to the evidence this is not frequent) we treat this as an obstacle rather than a hazard – the workflow
cannot be allowed to continue as defined. While an obstacle is a trigger introduced by the exception manager when something went wrong in the guideline, the hazard is a trigger introduced when a possible threat to the patient’s safety is detected by the exception manager. After an obstacle is detected the exception manager activates a goal to manage the error and recover nominal execution of the guideline, after a hazard is detected the exception manager activates a goal to concurrently monitor the patient state in order to find out if the detected hazard represents a real harm for the patient or is only a threat.

The exception manager comprises a hierarchical catalog of state-based exceptions and a run-time monitoring system that checks the state of running CIGs to recognize states corresponding to exceptions for which handlers exist in the catalog. In the catalog, each exception is associated with a goal that describes, as in [15], a state to be achieved, maintained, prevented, or ceased in order to manage the exception. The hierarchical specification of exceptions is based on the principle that to refine an exception E1 into a sub-exception E2, the state described by E2 should imply the state described by E1. For example, we can define a generic exception E1 with state specifying that a general practitioner, GP, is not authorized to prescribe a chemotherapy drug. E1 can be refined into a more specific exception E2 that characterizes the safety-based reason for withholding the prescription; E2’s state specifies that a GP is not authorized to prescribe a chemotherapy drug because there is evidence that the drug can have strong adverse effects on the white blood cell count and only oncologists can prescribe it.

The exception manager uses the following principles for selecting the exceptions to be fired at run-time. First, if exceptions E1 and E2 can be triggered, and E2 is a refinement of E1, then E2 is chosen over E1. The rationale for this is that the exception manager should prefer to trigger the most refined exceptions, which provide more detail concerning the detected medical error.
Second, if the conditions of E1 and E2 are not related by a refinement relationship then both E1 and E2 are chosen because they correspond to presumably different medical errors. For example, when a GP tries to prescribe a drug such as ACE Inhibitor to a pregnant woman who is taking potassium-sparing diuretics, two exceptions arise: E1, specifying that the drug (ACE Inhibitor) is not compatible with other drugs the patient is taking (potassium-sparing diuretic), and E2, specifying that the drug is contraindicated for pregnant women.

After the exception manager has selected the exceptions to trigger, it initializes the goals associated with the selected exceptions. Then, plans that match these goals can be selected to handle the exception.

State-based exceptions encapsulate medical errors that are detected. Formally, an exception is defined as the tuple:

\[
\langle \text{exceptionType}, \text{template}, \text{name}, \text{parameters}, \text{condition}, \text{parentException}, \text{triggeredGoal} \rangle
\]

\textit{triggeredGoal} where:

- \textit{exceptionType}: can be \textit{hazard} or \textit{obstacle}. Different subtypes of hazards and obstacles may be defined by specifying a value for the \textit{template} attribute (see below).

- \textit{Template}: as in [18] this determines the mode of execution of candidate plans to handle the exception, and can take the values \textit{parallelExecuting}, \textit{suspending(keystones)}, or \textit{discarding(keystones)}. In the case of the \textit{parallelExecuting} template, when the exception becomes InProgress it does not change the state of any keystone. By definition, hazards are always parallel executing exceptions (if discarding or suspension of tasks is required, the situation is an obstacle not a hazard). For a \textit{discarding(keystones)} template the goals and tasks indicated by the set \textit{keystones} change their state to Discarded after the exception is triggered (see figure 1). In the case of \textit{suspending(keystones)} the goals and tasks from the set
keystones change their state to Suspended after the exception is triggered. The keystones that were Suspended or Discarded can go back to the state they held at the time the exception was triggered only after the exceptional flow triggered to manage the exception is Completed.

- **Name, Parameters:** have the same interpretation as in keystones.
- **Condition:** corresponds to the state that needs to be satisfied in order to trigger the exception.
- **ParentException:** corresponds to the name of an existing exception. It allows hierarchies of exceptions from more abstract and generic to more concrete and refined to be defined.
- **TriggeredGoal:** once an exception is recognized, an exceptional flow should be activated to deal with the exception. The attribute triggeredGoal determines the goal that is triggered to manage the exception

### III. AN EXAMPLE OF WORKFLOW SPECIFICATION USING THE FRAMEWORK

In this section we present an example of a generic workflow for chronic diseases. A number of clinical guidelines for managing chronic diseases such as hypertension [10, 19] and lipid management [20] have a similar fundamental structure which can be abstracted as a common high-level guideline. As well as the generic workflow itself we mention some details related to its instantiation for the chronic disease hypertension.

**Workflow for managing chronic diseases**

The workflow is defined as a plan *Pattern_chronic_disease*, which introduces the goal
maintain_chronic_disease_managed (Table 1). This goal corresponds to periodically assessing the patient's state and, depending on the assessment, referring the patient to a specialist or prescribing a treatment (initial drug treatment, change of drug treatment or proposed lifestyle changes) and continuing the periodical assessment. A possible candidate plan for achieving the goal maintain_chronic_disease_managed is the plan Plan_management_chronic_disease (Figure 2). The plan starts with the goal achieve_assessment_done (Table 2). This goal should take on the state InProgress first at the time defined by its startTime parameter, and then again at regular intervals given by the frequency parameter. The goal is achieved when the patient has been assessed and the assessment has been added to the patient record. It must be achieved each time it becomes InProgress. The goal is aborted when the time to perform the assessment has expired. This can happen if the goal does not become InProgress for the first time as scheduled at the startTime, or if the required frequency is not respected on subsequent occasions when it becomes InProgress.

As shown in Figure 2 the goal achieve_assessment_done is repeated periodically for the purpose of monitoring. If the patient has a non-desirable state in the context of the specific disease under consideration then the goal achieve_treatment_chosen (Table 3) becomes InProgress, but if the state is a desirable one then the activation of the goal achieve_treatment_chosen is optional. For instance, if the state of the patient has significantly improved, a clinician can decide to reduce the drug's dose or keep the same treatment and continue monitoring the patient.

We use the function CheckIfDesirableState to provide a declarative definition of what a desirable state is for a disease. This function takes as input parameters the disease and the patient's state and it returns in a Boolean value the result of checking if the patient’s state is a desirable or not for the considered disease. For instance, for the chronic disease hypertension, the function
CheckIfDesirableState is formally defined as:

Function CheckIfDesirableState(hypertension, (sbp, dbp, setComorbidities)) ==

\[
\begin{cases}
\text{If } \left( \text{setComorbidities.Contains(kidney_disease)} \, || \, \text{setComorbidities.Contains(diabetes)} \right) \\
\text{&& IsInInterval(sbp,100,130) && IsInInterval(dbp,65,80)} \\
\text{||} \\
\left( \text{not setComorbidities.Contains(kidney_disease)} \, || \, \text{setComorbidities.Contains(diabetes)} \right) \\
\text{&& IsInInterval(sbp,100,140) && IsInInterval(dbp,65,90)}
\end{cases}
\]

return true else return false;

The above function takes as argument the disease (hypertension) and the results of patient assessment (sbp, dbp, setComorbidities), where sbp is the measured systolic blood pressure, dbp is the measured diastolic blood pressure and setComorbidities is the set of the patient’s comorbidities. The function considers the patient’s assessment to be desirable in the context of hypertension if the following condition holds: (1) the patient has kidney disease or diabetes and (sbp is in the range of values [100,130] and dbp is in the range of values [65, 80]), or (2) the patient does not suffer from kidney disease or diabetes and (sbp is in the interval [100,140] and sbp is in the interval [65, 90]). Otherwise the patient’s assessment corresponds to an undesirable state.

The goal achieve_treatment_chosen is considered Completed when the success condition becomes true, i.e., when the patient record has been updated with a prescribed treatment or with a reference to another actor. Possible treatments include: prescribing lifestyle changes such as dieting, exercising, dieting plus exercising, or prescribing drugs with or without lifestyle changes.

After the goal achieve_treatment_chosen becomes Completed, the initial goal maintain_
chronic_disease_managed (Table 1) is still InProgress, therefore the goal achieve_
assessment_done (Table 2) becomes InProgress again. A possible candidate plan for the goal
achieve_treatment_chosen is given by the plan Plan_choose_treatment, presented in Figure 3.
This plan attempts to achieve the goal by achieving one of three subgoals:
1. Goal achieve_drugs_prescribed: the goal is successfully achieved when drugs are
   prescribed.
2. Goal achieve_patient_referred: the goal is successfully achieved when the actor responsible
   for the patient decides to refer the patient to another actor, such as a specialist.
3. Goal achieve_TLS_changes_prescribed: the goal is successfully achieved by prescribing
   lifestyle changes, such as physical exercise, diet, etc.
As shown in Figure 3, if one of the subgoals 1, 2, or 3 is chosen and Completed then the goal
achieve_treatment_chosen is Completed. If subgoal 3, achieve TLS_changes_prescribed, is
chosen and Completed then there are two possibilities:
1. The workflow is Completed immediately.
2. Subgoal 1 or 2, not both, can be selected and after that subgoal is Completed the
   workflow is Completed.
The first subgoal in Figure 3, achieve_drugs_prescribed is achieved when the actor has chosen a
treatment from a set of drug treatments suggested by the decision-support system, based on the
specific-patient data. The treatments suggested by the support system correspond to
combinations of drugs that are not contraindicated, are compelling drugs, or are good drug
partners to current medications.
IV EXCEPTIONS IN THE EXAMPLE WORKFLOW

A. Properties to be satisfied by a workflow for managing chronic diseases

Below we propose a list of desirable properties that should be satisfied by any workflow for managing chronic disease, and we analyze whether the workflow of section III satisfies them or some exception should be specified:

1. The goal of managing a chronic disease should persist until a dramatic exception occurs, such as the death of the patient, the patient abandoning treatment, etc. In section IV.B.3 we propose an exception to discard the goal maintain_chronic_diseaseManaged when the patient dies.

2. If the patient's state is not considered desirable in the context of the disease under consideration the patient will eventually be referred, prescribed a drug treatment, or prescribed a lifestyle change. In our workflow there is the danger that a patient is repeatedly referred to a specialist without being prescribed any treatment. In section IV.B.5 we propose the exception referral_policy_unsatisfied for limiting referrals.

3. If the patient state is desirable in the context of the disease under consideration a different treatment can nonetheless be prescribed. For instance, some drugs can be eliminated or their dose can be reduced, a clinician can propose lifestyle changes, or even refer the patient to another physician. From the way we defined the goal achieve_treatment_chosen we can be sure that our workflow satisfies this property.

B. An example of an exception-based manager using the framework

Below we provide some examples of exceptions that can populate the catalog of exceptions for the workflow presented in section III. The examples given here are specified at a generic level,
so that they can be applied to a wide range of specific workflows regardless of the details of a particular instantiation for a specific disease.

Examples of Obstacles:

1. A goal is **InProgress**, a **candidatePlan has been chosen and Completed**, but the goal's **successCondition is still not satisfied**: when suspending obstacle *unachieved_goal_despite_candidate_completed* (Table 4) occurs the goal that is unachieved is Suspended until the reason for the failure to achieve it is found. This obstacle triggers the goal *achieve_reason_goal_unachieved_found* which takes as input parameters: a reference to the actor who is responsible for performing the Suspended goal (and is thus responsible for finding the reason why the goal was not achieved), and the Suspended goal itself. For instance, the goal *achieve_drug_treatment_chosen* can be in this obstacle state because, for some reason, no treatment could be prescribed or the patient could not be referred.

2. A keystone is **Discarded** because its **abortCondition is satisfied**: when the obstacle *is_discarded* (Table 5) occurs the keystone is Discarded and we want to find the reason behind the discarding event. Therefore, this obstacle triggers the goal *achieve_reason_is_discarded_found* with the following parameters: a reference to the actor who is performing the Discarded keystone (and who is thus responsible for finding the reason why the keystone was Discarded), and the Discarded keystone itself. In the hypertension example a task may be discarded, for example, if we want to assess the patient's blood pressure (goal *achieve_assessment_done*, Table 2), and measuring the blood pressure (candidate plan *Plan_hypertension_assessment*, Table 6) cannot be done on time (given by parameter *startTime*). In this case the exception-manager triggers the discarding obstacle *is_discarded.*
3. **A keystone is Discarded when it refers to a patient who has died:** the obstacle *patient_dies* discards the keystones that have as parameter the patient that died and it triggers the goal *achieve_communicate_patient_death* to provide the user with the reasons for discarding the keystone.

4. **Obstacle refinement:** the discarding obstacle *is_discarded* (Table 5) is very generic and can be refined into more concrete sub obstacles. For example, it can be refined into *time_expired* (Table 7) whose *condition* corresponds to the state where the keystone did not start at *startAt* or did not cycle with the frequency described by *cycleInterval*. This obstacle triggers the goal *achieve_rescheduled* with parameters: the reference to the actor who is responsible for the Discarded keystone (and thus for rescheduling it), the keystone that needs to be rescheduled, the new starting time and the new frequency for the keystone. The goal *achieve_rescheduled* succeeds when the *startAt* and/or *frequency* attributes of the keystone have been changed in order to start it before the next iteration was originally scheduled or to increase its frequency of iteration.

The discarding obstacle *time_expired* can be refined into the discarding obstacle *not_available_resource* (Table 8). Its *condition* specifies that the time violation occurred because a resource which was needed to perform the task or goal was not available. For instance, the *Plan_hypertension_assessment* (Table 6) can be Discarded when measurement of sbp and dbp was not carried out on time (for example because the measurement device was unavailable or broken). This obstacle triggers the goal *achieve_rescheduled*, explained above.

5. **A particular goal is Discarded:** with the discarding obstacle *referral_policy_unsatisfied* we can discard the goal *achieve_patient_referred* when the function *SatisfyReferralPolicy* determines that the institution’s policy for referrals has been violated. This obstacle triggers the
goal achieve_communicate_exception to explain to the actor responsible for goal achieve_patient_referred the reasons why the patient cannot be referred.

Examples of Hazards

6. An undesirable patient state has been assessed, the reason is unknown: the parallel executing hazard undesirable_patient_state_after_drug_treatment (Table 9) is triggered when the patient is under drug treatment for a certain disease and the results of the last assessment show that the patient’s progression is abnormal. This hazard is very generic and is triggered when there is no information to explain the reasons for the patient’s undesirable state. For example, when an abnormal laboratory test has been detected soon after starting a drug (e.g., the white blood cell count was low after starting 5-Fluorouracil treatment), but there is no detailed knowledge in the knowledge-base about which drug is associated with which abnormal laboratory test, the hazard triggers the goal achieve_responsible_physician_alerted so that the physician who is responsible for the patient’s drug prescription can decide if this is a short-term reaction to the drug treatment that warrants follow up or immediate alteration of the drug.

7. The selected task is not the most recommended one: the parallel executing hazard not_most_recommended_choice triggers the goal achieve_rescheduled explained in IV.B.4. For instance, if the goal achieve_drug_treatment_chosen has been achieved by selecting a drug-treatment that is not the most recommended one, then the exception manager should trigger this hazard. Based on [13], this hazard could be refined into more concrete hazards, for example: the time constraints of the chosen task are not the most recommended for achieving efficacy and least harm (sequencing), the chosen task is not the most recommended for ensuring the overall plan’s efficacy (efficacy), the chosen task can undermine the benefits of other active tasks
(diminution), the chosen task can exacerbate states categorized as hazardous (exacerbation), etc.

V DISCUSSION

The framework presented above allows the modeling of exceptions that occur during CIG execution and their handling, and addresses some important barriers to CIG adoption and implementation that have not been addressed previously. One such barrier is the ability to provide decision-support in the case of exceptions. While most guideline-based decision-support systems would fail completely when an exception occurs, our framework allows exceptions to be detected and handled so that the system can recover and offer some advice. A second barrier is that most guideline-based decision-support systems do not work well when the encoded knowledge is incomplete. However, the hierarchical exception-handling framework outlined above can provide decision support appropriate to the type of exception even if detailed information about specific exceptions has not been encoded (as in the example of section IV.B.6).

Application of the framework

We see four particular benefits that this approach can offer in healthcare informatics applications. Firstly, the use of goals allows a modular separation between the essence of the medical guidelines and the possible ways to achieve them by the enactment of concrete workflows during run-time. The fact that workflows in our approach are goal-based and modular confers several related benefits [9]: goal-based clinical workflows may be customized at run-time. This is important because healthcare environments are highly dynamic and adapting a conventional clinical guideline to work in a new location can require significant effort [21]. A modular approach also allows tasks within the workflow to be distributed and delegated across arbitrary
actors within a clinical team.

Secondly, our approach of identifying reusable generic patterns of clinical workflow and exception types, and our use of a hierarchical exception catalog, allows us to abstract, reuse and share lessons learned from the specification of concrete scenarios (medical guidelines, medical error types, and strategies to deal with and recover from medical errors). Exceptions are state-based, so they are defined at design time in terms of possible states of error and are not associated at design time to particular points in the workflow. Only during run-time does the exception-manager monitor the state of the workflow to discover at which points of the workflow exceptions should be triggered. This systemic approach allows the specification of the catalog of exceptions independently of the workflow.

Thirdly, our classification of exceptions as obstacles or hazards, and our classification of recovery strategies as discarding, suspending, or parallelExecuting, based on how harmful an exception can be for the patient, provides a principled basis for automated or semi-automated re-planning of workflow on-the-fly. While obstacles have been considered before in the context of strategies for recovery and repair of workflow [11,17], as far as we know, our approach based on the definition of hazards for closely monitoring situations to prevent future errors is novel.

Fourthly, our framework provides a basis for interoperability between different applications and guideline formalisms; while the exception handling library has to be expressed in our workflow-based formalism, plans defined in different CIG languages, such as Asbru or PROforma, could be proposed at run time to meet these goals. For example given the plan from Figure 2 with goals achieve_assessment_done and achieve_treatment_chosen, at run time a PROforma plan with attribute goal=achieve_assessment_done might be enacted to achieve the first goal, while an Asgaard-Asbru [8] plan with attribute intention=achieve_treatment_chosen might be enacted.
for the second goal. There is no restriction over the type of applications or IT tools that can be referenced by an automatic task (see Figure 4-g). This approach to interoperability between CIGs defined in different languages contrasts with alternative approaches based on common interchange language standards for sharing CIGs, like the GuideLine Interchange Format GLIF in [22].

To take full advantage of this possibility our framework should be complemented with:

1. An ontology mapping system for matching virtual data to concrete data in the corresponding application or technology used to implement an automatic task. For example a run-time ontology mapping should associate formal parameters with the actual variables used in the hospital where the workflow is running.

2. Interfaces for exchanging input and output parameters between the framework and the applications that implement automatic tasks.

**Implementation of the framework**

In section II we have introduced a framework for the specification of process and exceptions (see Figure 4-a). In sections III and IV we have explained how to use the framework to specify processes (see Figure 4-b) and catalogs of exceptions (see Figure 4-c) at design time. Although we have not implemented the process enactment engine (see Figure 4-d) and the exception manager (see Figure 4-e), we have provided formal semantics for them based on a labeled state transition system for keystones (section II.A) and rules for selection of exceptions during run-time (section II.B).

While generic exceptions can be defined independently of process definitions, some refined exceptions need to refer to concrete process definitions. Recent research on domain-independent exceptions [23] has confirmed that it is possible to define generic hierarchical catalogs of
exceptions that are both robust enough to be reused in different workflows and also internally self-consistent. In our approach the process enactment engine (see Figure 4-d) and the exception manager (See Figure 4-f) create instances of processes and exceptions at run time based on definitions provided at design time. Therefore, while the process enactment engine and the exception manager may be implemented quite separately, they must communicate at run time: the exception manager needs to know the state of the system in order to choose which exceptions should be activated, and the enactment engine needs to know if any goal has been triggered by active exceptions.

In this study, we have used an object-oriented pseudo-code for writing clinical expressions for goals, tasks, hazards, and obstacles. This pseudo-code can be easily mapped into an object-oriented language like JAVA. Neither have we included specific proposals for linkage to EMRs and clinical information systems (such as order entry systems). Good candidate standards for supporting implementation are HL7’s Object-oriented Guideline Expression Language (GELLO) [24] and HL7’s Virtual Medical Record (vMR) model. The Clinical Decision Support Technical Committee is currently developing the vMR on the basis of the HL7 CareRecord model. Compilers for GELLO that work with the vMR are currently under development by HL7 members.

**Related work**

Goal-based reasoning is an approach that has been in use in clinical rule-based systems since the 1960's. The most famous early decision-support system that used goal-based reasoning is Mycin [25]. Mycin was used to provide diagnosis and treatment recommendations about infectious diseases. Mycin tries to diagnose the organism(s) that caused the infection by applying backward chaining over production rules. Mycin tries to prove the consequent of the rules (e.g., that E. Coli
caused the infection) by proving the antecedent conditions (e.g., Gram stain is negative, the patient is a compromised host) by either asking for information from the user or using other production rules to prove the existence of these conditions.

Goal-based specification of clinical workflow is not a new approach. Scenario-based \([10]\) notions of goals have been proposed by Tu and Musen \([26]\). In SAGE \([26]\) they present a formal framework to specify generic workflows at design time that can be specialized at run time based on the concept of a scenario \([10]\): a particular patient state at a specific point in the enacted CIG, workflow contexts or triggering events. A workflow context is a patient care event, defined by an agent in a role performing an act on an object for a patient in a setting. Other events that trigger intervention may be guideline actions or clock events that generate periodic population-based surveillance.

Similarly in GLIF3 \([27]\), patient state steps can be used for several purposes, including labeling the patient state at important points in the guideline flow or as entry/exit points into the guideline. When patient state steps are used to mark goal states (i.e., exit states) the implication is that if the GLIF3 algorithm was followed then the guideline reaches the goal patient state.

In PROforma \([16, 17]\) a task can be associated with a goal, specified in the Red Representation Language (R\(^2\)L) \([14]\), a time oriented knowledge representation language. Before execution predicates in the R\(^2\)L language are translated into another language, called L\(_{R\^2L}\) (Logic of R\(^2\)L), a language based on predicate logic. In PROforma, as in GLIF3, the interpretation of a goal is that it will be achieved when the task to which it is associated is successfully completed. As with GLIF3, in current PROforma implementation this is not formally enforced and goals therefore act as informal annotations on tasks.

The Asgaard-Asbru \([8]\) project implements an idea similar to ours but goals and tasks are not
given the status of first order objects, as we do here. In Asgaard, workflows are defined in terms
of tasks and the tasks have an attribute that specifies the goal to achieve. But their notion of goal
is broader than ours; for them, goals are defined as achieving, maintaining, or avoiding actions or
temporal patterns of state variables, whereas in our approach, goals only represent temporal
patterns of state variables that should be achieved, maintained, prevented or ceased (e.g., goal of
achieving systolic blood pressure of less than 120 mmHg starting after the patient was assessed
with high blood pressure). Like Asgaard our framework allows a plan to be chosen at run time to
achieve a goal, provided that there is evidence that the goal is achievable by the enactment of the
chosen plan. But their framework goes further allowing the user to choose a non recommended
plan to achieve a goal. If after plan execution the goal is not achieved this is recorded for further
inspection, known as critiquing the guideline enactment.

With respect to the specification of exception handlers, our notion of exception is based on a
similar state-based notion from the PROforma language [17]. But in our approach exceptions
can be hierarchically organized and can be classified between discarding or suspending obstacles
and parallel executing hazards.

Like us, Tu and Musen [26] differentiate between normal flows and exceptional flows and they
contemplate the specification of scenario-based exception handlers, but their exception catalog is
not hierarchical.

Our idea that exceptions should be specialized, reused, and proposed from the state-based
attributes of the procedural guideline specification is shared with KAOS [28], a generic goal-
directed requirement engineering approach. Besides in our approach events can trigger goals,
exceptions, or tasks when certain conditions hold. This is an extension of the well-known Event
Condition Action (ECA) paradigm [29] where events conditionally trigger actions.
Future work

We share with Tu and Musen the idea that great benefit to Health Informatics will accrue from a well populated repository of generic workflows or patterns that can be proposed, studied, shared, reused, and specialized by the medical community using a framework, such as the one they propose in [10, 30] or the one we propose here. We also believe that safer CIGs will result if the healthcare community is able to share the lessons learned on medical errors and good practices to reduce and prevent errors. Using our approach it would be possible to propose patterns to capture generic normal workflows corresponding to general good medical practices, and also provide generic catalogs of exceptions to deal with exceptional workflows for capturing and managing medical errors and monitoring hazardous situations that could harm the patient to prevent future errors. For instance, we have presented the plan Pattern_chronic_disease that could be suggested as a normal flow pattern for managing chronic diseases. In section IV we have proved that the proposed workflow satisfies critical properties that a pattern for specifying chronic diseases should satisfy. We have also shown, in section IV.B, that our framework can be used to provide the proposed patterns with safety controls to capture and manage medical errors. This framework is state-based and systemic and could be used to specify global errors that arise not from specific points in the workflow but from the interleaving execution of multiple workflows. So far we have been using the framework to specify errors arising from the execution of a single workflow but we expect to study in the future the suitability of the framework for scenarios corresponding to patients undertaking parallel treatments. We expect to tackle exceptions arising from:

- Relationships between tasks: the concurrent execution of task A (prescribe ACEI) in guideline G1 (hypertension) and task B (stop taking ACEI) in guideline G2 (chronic cough)
produces an exception E (conflicting tasks). A goal triggered by E might determine whether alternative plans can be used to achieve goals G1 and G2.

- Relationships between goals: for some patients two active goals may conflict. For instance for a patient undergoing fertility treatment when diagnosed with ovarian cancer the goals [achieve_cancer_removed] and [achieve_become_pregnant] are in conflict.
- The global effect of new local knowledge: new knowledge acquired during the execution of one workflow can impact other active workflows and introduce errors or hazards. For instance if during a routine assessment it is discovered that the patient, who is under drug treatment, is pregnant then an exception may be triggered to avoid medication which could affect her pregnancy.

VI CONCLUSIONS

We have introduced a framework for the specification of process and exceptions that is goal-based, modular, systemic, extensible and hierarchical.

Although we have not implemented the process enactment engine and the exception manager, we have provided formal semantics for them and rules for selection of exceptions at run-time.

The future implementation of this approach will help to evaluate the reusability and the degree of interoperability allowed by the framework proposed here and most importantly if medical errors arising from abnormal or undesirable unpredictable situations could be reduced with the incorporation of a goal-based exception manager.
REFERENCES


Workshop on Requirements for High Assurance Systems; Essen; 2002.


**Figure and Table Legends**

Figure 1: Transition system describing the possible states that a keystone can take in the framework.

Figure 2: *Plan_management_chronic_disease*, a candidate plan for the goal *maintain_chronic_disease_managed*. Circles denote start and end points, hexagons denote goals. See text for details.

Figure 3. *Plan_choose_treatment*, a candidate plan for the goal *achieve_treatment_chosen*. Triangles denote XOR or AND split points.

Figure 4: Schematic representation of the dependency relations between components needed to implement the framework.

Table 1: goal *maintain_chronic_disease_managed*

Table 2: goal *achieve_assessment_done*

Table 3: goal *achieve_treatment_chosen*

Table 4: suspending obstacle *unachieved_goal_despite_candidate_completed*

Table 5: discarding obstacle *is_discarded*

Table 6: plan *Plan_hypertension_assessment*

Table 7: discarding obstacle *time_expired*

Table 8: discarding obstacle *not_available_resource*

Table 9: hazard *undesirable_patient_state_after_drug_treatment*
Table 1: Goal `maintain_chronic_disease_managed`

<table>
<thead>
<tr>
<th>name</th>
<th>maintain_chronic_disease_managed</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameters</td>
<td>startTime, frequency, disease, patientID, records$^1$</td>
</tr>
<tr>
<td>precondition</td>
<td>isChronic(disease)</td>
</tr>
<tr>
<td>startAt</td>
<td>startTime</td>
</tr>
<tr>
<td>successCondition</td>
<td>False</td>
</tr>
</tbody>
</table>

| invariantCondition$^2$ | \[
|------------------------|---------------------------------
| AssessmentsAsScheduled(patientID, disease, records, startTime, frequency) \&\&

where desirableState == CheckIfDesirableState (disease, assessment.GetValue())

$^1$ The `Records` parameter is of type `RecordType`, which is defined in terms of the attributes: `patientID` (unique patient identifier), `disease` (name of a disease to which the record is related), `Actclass` (with possible values assessment, treatment, referral), `value` (list of pairs (variable, value) to describe the assessment or treatment, for instance (sbp, sbpvalue)(dbp, dbpvalue) for the assessment of the hypertension disease), `timeStamp` (corresponding to the time the variable has been assigned a value). Only if the record has attribute `Actclass == referral` then an additional attribute `referringActor` is used to indicate the actor who is referring the patient to a new act or specified by the additional attribute `referredToActor`. If the record has attribute `Actclass == treatment` then an extra attribute `type` describes the type of treatment, taking the values: drugs, diet, exercise, diet and exercise, drugs and diet, drugs and exercise, or drugs and diet and exercise.

$^2$ The function `AssessmentsAsScheduled` returns true if the assessment took place as scheduled: first assessment at `startTime`, and further iteration of the assessment with the required `frequency`. The function `RetrieveRecord` returns the list of medical records of type `RecordType` with patient identifier `patientID`, for the `disease` and `Actclass` specified as input parameters. From the retrieved records, only the last record is considered. Function `CheckIfDesirableState` (explained in Section III.A) returns true if the most recent assessment of the patient's state is within the desirable values for the disease. Therefore 1) the assessment should be under scheduled and two options are possible: 2.a) it is the first assessment or 2.b) it is not the first assessment. If it is not the first assessment and if it is not a desirable state then: (2.b.i) the patient has been prescribed a treatment after the assessment, or (2.b.ii) he has been referred after the last assessment.
### Table 2: goal achieve_assessment_done

<table>
<thead>
<tr>
<th>Goal Type</th>
<th>Achieve</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>achieve_assessment_done</td>
</tr>
<tr>
<td>Parameters</td>
<td>startTime, frequency, patientID, disease, records</td>
</tr>
</tbody>
</table>
| Precondition | \[
|             | records.RetrieveRecord(patientID, disease, assessment).RetrieveLast() == assessment &
|             | (assessment == null &
|             | now() == this.GetstartAt() ) \ || 1
|             | assessment != null &
|             | IsinInterval(now(), assessment.GetTimeStamp(),
|             | assessment.GetTimeStamp() + frequency ) \ || 2 |
| StartAt      | startTime |
| Cycle Interval| Frequency |
| Success Condition | records.RetrieveRecord(patientID, disease, assessment).RetrieveLast() == newAssessment
|             | && newAssessment != null &&
|             | newAssessment.GetTimeStamp() > assessment.GetTimeStamp() |
| Abort Condition | This.getState() == Dorman &
|               | records.RetrieveRecord(patientID, disease, assessment).RetrieveLast() == assessment &
|               | (assessment == null &
|               | now() == this.GetstartAt() ) \ || 1
|               | assessment != null &
|               | not ( IsinInterval(now(), assessment.GetTimeStamp(),
|               | assessment.GetTimeStamp() + frequency ) ) \ || 2 |

3 (1) it is the time (now()) that the goal should be inProgress for first time (startAt), or (2) it is the time the goal should cycle, considering that it is not the first time it is inProgress. Function IsInterval takes as argument three time stamps and returns true if the first parameter p1 is in the interval of time determined by the second and third parameters [p2,p3].

4 A new assessment has been done and added to the medical records

5 The time constraints for doing the assessment were not satisfied because (1) the first assessment was not started as specified at startAt, or (2) no new assessment was performed with the periodicity specified by cycleInterval.
Table 3: goal achieve_treatment_chosen

<table>
<thead>
<tr>
<th>goalType</th>
<th>Achieve</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>achieve_treatment_chosen</td>
</tr>
<tr>
<td>Parameters</td>
<td>disease, patientID, records, startTime</td>
</tr>
<tr>
<td>startAt</td>
<td>startTime</td>
</tr>
<tr>
<td>Precondition</td>
<td>now()==this.GetStartAt()&amp;&amp;</td>
</tr>
<tr>
<td></td>
<td>records.RetrieveRecord(patientID,disease,assessment).RetrieveLast()==assessment&amp;&amp;</td>
</tr>
<tr>
<td></td>
<td>assessment!=null</td>
</tr>
<tr>
<td>Actor</td>
<td>initialActor</td>
</tr>
</tbody>
</table>
| successCondition | records.RetrieveRecord(patientID,disease,treatment).RetrieveLast()==treatment &&
|                | treatment.GetTimeStamp()==now() ||
|                | records.RetrieveRecord(patientID,disease,referral).RetrieveLast()==referral&& |
|                | referral.GetReferringActor()== initialActor && |
|                | referral.GetReferredToActor() != initialActor |
| abortCondition | This.getState()==Dormant && now()> this.GetStartAt() |

---

6 An assessment has been performed before and the time set for starting this goal has arrived.
7 (1) the patient record is updated with a treatment or (2) a referral to another actor is carried out.
<table>
<thead>
<tr>
<th>exceptionType</th>
<th>Obstacle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template</td>
<td>Suspending({goal})</td>
</tr>
<tr>
<td>Name</td>
<td>unachieved_goal_despite_candidate_completed</td>
</tr>
<tr>
<td>Parameters</td>
<td>goal, plan</td>
</tr>
<tr>
<td>Condition&lt;sup&gt;8&lt;/sup&gt;</td>
<td>goal.GetState()==InProgress &amp; goal.GetCandidatePlans().Contains(plan) &amp; plan.GetState()==Completed &amp; CheckSatisfaction(goal.GetSuccessCondition(), goal.GetValues ()==false</td>
</tr>
<tr>
<td>parentException</td>
<td>null</td>
</tr>
<tr>
<td>triggeredGoal</td>
<td>achieve_reason_goal_unachieved_found(goal.GetActor(),goal)</td>
</tr>
</tbody>
</table>

<sup>8</sup> The goal is inProgress, a candidate plan for the goal has been Completed but the goal’s successCondition is not satisfied. Function CheckSatisfaction evaluates the goal’s successCondition replacing the variables for the corresponding values.
**Table 5: Discarding obstacle *is_discarded***

<table>
<thead>
<tr>
<th>exceptionType</th>
<th>Obstacle</th>
</tr>
</thead>
<tbody>
<tr>
<td>template</td>
<td>Discarding({keystone})</td>
</tr>
<tr>
<td>Name</td>
<td>is_discarded</td>
</tr>
<tr>
<td>Parameters</td>
<td>keystone</td>
</tr>
<tr>
<td>Condition</td>
<td>CheckSatisfaction(keystone.GetAbortCondition(), keystone.GetParameters())==true</td>
</tr>
<tr>
<td>parentException</td>
<td>Null</td>
</tr>
<tr>
<td>triggeredGoal</td>
<td>Achieve_reason_is_discarded_found(keystone.GetActor(),keystone)</td>
</tr>
<tr>
<td>name</td>
<td>Plan_hypertension_assessment</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>parameters</td>
<td>startTime, patientID, disease, records, dbp, sbp, setComorbidities</td>
</tr>
<tr>
<td>precondition</td>
<td>disease==hypertension</td>
</tr>
<tr>
<td>startAt</td>
<td>StartTime</td>
</tr>
<tr>
<td>goalsAchievable</td>
<td>achieve_assessment_done</td>
</tr>
</tbody>
</table>
| successCondition | \[
\begin{align*}
\text{records.RetrieveRecord(patientID,disease,assessment).RetrieveLast()==assessment} \\
\text{&\& assessment.GetTimeStamp()==now()} &\& \\
\text{assessment.ObtainSbp(sbp)} &\& assessment.ObtainDbp(dbp) &\& \\
\text{assessment.ObtainSetComorbidities(setComorbidities)}
\end{align*}
\] |
| abortCondition | Now()>startTime |

---

\( (1) \) the assessment for the disease hypertension is completed and \( (2) \) the patient’s systolic blood pressure obtained during the assessment is saved in variable \textit{sbp}, the diastolic blood pressure is saved in variable \textit{dbp}, and the set of comorbidities related to hypertension (diabetes, kidney disease) is saved in the variable \textit{setComorbidities}. 

---

41
### Table 7: Discarding obstacle time_expired

<table>
<thead>
<tr>
<th>exceptionType</th>
<th>Obstacle</th>
</tr>
</thead>
<tbody>
<tr>
<td>template</td>
<td>Discarding({keystone})</td>
</tr>
<tr>
<td>name</td>
<td>time_expired</td>
</tr>
<tr>
<td>parameters</td>
<td>keystone, schedule</td>
</tr>
<tr>
<td>condition&lt;sup&gt;10&lt;/sup&gt;</td>
<td></td>
</tr>
</tbody>
</table>
|                 | \[
|                 | 1. keystone.ObtainCompletionTime(completionTime)&&(completionTime==null &\& now()>keystone.GetStartAt())
|                 | 2. ||
|                 | 3. completionTime!=null &\&keystone.GetCycleInterval()!={}now()>completionTime+keystone.GetCycleInterval() |
| parentException | is_discarded                   |
| triggeredGoal   | Achieve_rescheduled(keystone.GetActor(), keystone, newStartTime, newFrequency) |

<sup>10</sup> (1) the last time the keystone was completed is saved in variable `completionTime` and: (2) it is the first time the keystone should become inProgress but the time to start the keystone has expired, or (3) it is not the first time the keystone is inProgress, it is a cyclic keystone, but the time to start the new cycle has expired.
### Table 8: Discarding obstacle *not_available_resource*

<table>
<thead>
<tr>
<th>exceptionType</th>
<th>Obstacle</th>
</tr>
</thead>
<tbody>
<tr>
<td>template</td>
<td>Discarding({keystone})</td>
</tr>
<tr>
<td>Name</td>
<td>not_available_resource</td>
</tr>
<tr>
<td>parameters</td>
<td>resource, keystone, schedule</td>
</tr>
<tr>
<td>condition(^{11})</td>
<td>keyston.ObtainCompletionTime (completionTime) &amp;&amp; (1)</td>
</tr>
<tr>
<td></td>
<td>(\text{completionTime} == \text{null} &amp;&amp; \text{now()} &gt; \text{keystone.GetStartAt()}) (2)</td>
</tr>
<tr>
<td></td>
<td>(\text{completionTime} == \text{null} &amp;&amp; \text{keystone.GetCycleInterval}() == \text{null} &amp;&amp; \text{now()} &gt; \text{completionTime} + \text{keystone.GetCycleInterval}()) (3)</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; \text{not(available(resource, now()))} (4)</td>
</tr>
<tr>
<td>parentException</td>
<td>time_expired</td>
</tr>
<tr>
<td>triggeredGoal</td>
<td>achieve_rescheduled(keystone.GetActor(), schedule, newStartTIme, newFrequency)</td>
</tr>
</tbody>
</table>

\(^{11}\) As in the obstacle *time_expired* (1) the last time the keystone was *completed* is saved in variable *completionTime* and: (2) it is the first time the keystone should become *inProgress* but the time to start the keystone has expired, or (3) it is not the first time the keystone is *inProgress*, it is a cyclic keystone, but the time to start the new cycle has expired. In addition, (4) the resource required is not available.
### Table 9: Hazard *undesirable_patient_state_after_drug_treatment*

<table>
<thead>
<tr>
<th>exceptionType</th>
<th>Hazard</th>
</tr>
</thead>
<tbody>
<tr>
<td>template</td>
<td>ParallelExecuting</td>
</tr>
<tr>
<td>name</td>
<td><em>undesirable_patient_state_after_drug_treatment</em></td>
</tr>
<tr>
<td>parameters</td>
<td>patientID, records, disease, responsibleTreatment</td>
</tr>
<tr>
<td>parentException</td>
<td>Null</td>
</tr>
<tr>
<td>triggeredGoal</td>
<td>achieve_responsible_alerted( patientID, disease, treatment, assessment, responsibleTreatment )</td>
</tr>
</tbody>
</table>