Chapter 10
Interactive Planning-Based Hypothesis Generation with LTS++

Shirin Sohrabi, Octavian Udrea, Anton Riabov, and Oktie Hassanzadeh

Abstract We present LTS++, an interactive development environment for planning-based hypothesis generation motivated by applications that require multiple hypotheses to be generated in order to reason about the observations. Our system uses expert knowledge and AI planning to reason about possibly incomplete, noisy, or inconsistent observations derived from data by a set of analytics, and generates plausible and consistent hypotheses about the state of the world. Planning-based reasoning is enabled by knowledge models obtained from domain experts that describe entities in the world, their states, and relationship to observations. To address the knowledge engineering challenge, we have developed a language, also called LTS++ that allows the domain expert to specify the state transition model and encoding of the observations without any knowledge of AI planning or existing planning languages (i.e., PDDL). LTS++ integrated development environment facilitates model testing and debugging, generating, and visualizing multiple hypotheses for user-provided observations, and supports model deployment for online observation processing, publishing generated hypotheses for analysis by experts or other systems. To compute hypotheses we use an efficient planner that finds a set of high-quality plans. We experimentally evaluate our planning algorithm and conduct empirical evaluation to demonstrate the feasibility of our approach and the benefits of using planning-based reasoning. In this chapter we focus on describing the modeling and the knowledge engineering challenges of our system.

S. Sohrabi (✉) · O. Udrea · O. Hassanzadeh
IBM Research, Yorktown Heights, NY, USA
e-mail: ssohrab@us.ibm.com; udrea@us.ibm.com; hassanzadeh@us.ibm.com

A. Riabov
Logitech Inc., Newark, CA, USA
e-mail: ariabov@logitech.com

M. Vallati, D. Kitchin (eds.), *Knowledge Engineering Tools and Techniques for AI Planning*, https://doi.org/10.1007/978-3-030-38561-3_10

1 Introduction and Motivation

The set of planning-based tools, collectively called LTS++, address the hypothesis generation problem that arises in applications that require multiple hypotheses to be generated in order to reason about possibly incomplete or inconsistent sequences of observations received from external sources. For example, when analyzing observations derived from sensor data in intensive care, the goal can be to generate plausible hypotheses about the condition of the patient. The resulting hypotheses can then be further refined and analyzed to create a recovery plan for the patient. In another application, decisions aimed to prevent malware spread in computer networks can be based on hypotheses about change in behavior of individual hosts generated by reasoning about observations of network traffic over time.

The core idea of the approach to planning-based hypothesis generation we implement in LTS++ is the following. Modeling the hypothesis generation problem as one of inferring a sequence of state transitions from a sequence of observations and transforming the sequence of observations together with the state transition model into a planning task. In particular, we extend the work of Sohrabi et al., [19] to address unreliable observations and generate multiple near-optimal lowest-cost plans, mapping the generated plans to hypotheses [17, 27]. This mapping ensures that lower cost plans are mapped to more plausible hypotheses; hence, finding a number of lowest-cost plans results in the same number of most plausible hypotheses.

Our LTS++ implementation uses an efficient planner that finds top-*k* plans, i.e., *k* plans such that no valid plans with lower cost exist [11, 16, 24]. We have evaluated several algorithms for this purpose, and currently use the *k*-shortest path algorithm K* [1]. More details can be found in [16].

Knowledge engineering requirements come to the forefront in designing a system like LTS++, where domain knowledge is encoded and maintained directly by the domain experts, such as clinicians or network security engineers. To address these requirements, we developed the LTS++ language that allows the domain experts to easily describe the state transition models and observations specific to their domain, without requiring the experts to learn about the underlying planning technologies or Planning Domain Definition Language (PDDL) [13]. The LTS++ browser-based Integrated Development Environment (IDE) includes an editor with syntax highlighting and static error checking, as well as integrated tools for interactive model testing and debugging, generating, and visualizing multiple hypotheses for user-provided observations. Models created in the IDE can then be deployed to LTS++ servers to generate hypotheses automatically as observations are received, generating alerts based on hypotheses for further analysis by experts or other systems.

We build upon a significant body of prior research. While expert judgment is the primary method used for generating hypotheses and evaluating their plausibility, automated methods have been proposed, to assist the expert, and help improve accuracy and scalability. Notably, model-based diagnosis methods can determine whether observations can be explained by a model (e.g., [3]). Also, several researchers have proposed use of automated planning technology to address several related classes of problems including diagnosis (e.g., [7, 18]), plan recognition (e.g., [14, 15, 22]), and finding excuses [5]. These problems share a common goal of finding a sequence of actions that can explain the set of observations given the model-based description of the system. However, most of the existing literature makes an assumption that the observations are reliable and should all be explainable according to the model. But that is not true in general; as a further complication, we cannot assume the system model is complete. The hypothesis generation approach we propose handles the unreliable observations and incomplete models by offering multiple alternative hypotheses explaining each given observation sequence. Our LTS++ tool automates the generation and evaluation of hypotheses in addition to addressing the knowledge engineering challenges of encoding and maintaining models.

While we have performed experimental evaluation and conducted empirical evaluation to demonstrate the feasibility of our approach and the benefits of using planning-based reasoning, in this chapter, we focus on describing the modeling and the knowledge engineering challenges of our system. In particular, in Sect. 2, we describe our two applications, early detection of complications in ICU and early detection of malware in computer networks. In Sect. 3, we describe the hypothesis generation problem and its relationship to planning. In Sect. 4, we describe our proposed language LTS++ and its main elements as well as its relationship to a planning problem. We will then discuss the LTS++ IDE and provide a number of example hypotheses in Sect. 5. We will conclude with a discussion of related work and summary.

2 Application Description

In this section we describe two real-world applications that motivate our approach: the early detection of patient complications in Intensive Care Units (ICUs) and suspicious behavior of hosts (computers) in computer networks. A key characteristic of these applications is that the true state of monitored patients, network hosts, or other entities, while essential for timely detection and prevention of critical conditions, is not directly observable. Furthermore, there are several ways of analyzing the raw data to create observations about the entity, and there are multiple potentially ambiguous observations, each of which can have differing interpretations. We must then analyze the sequence of available observations to reconstruct or estimate the entity state, and use that to drive further analysis or take specific actions. To make this possible, our approach relies on a model of the entity consisting of states, transitions between states, and a many-to-many correspondence between states, observations, and actions. The model is a representation of the knowledge a domain expert uses to perform the corresponding monitoring and

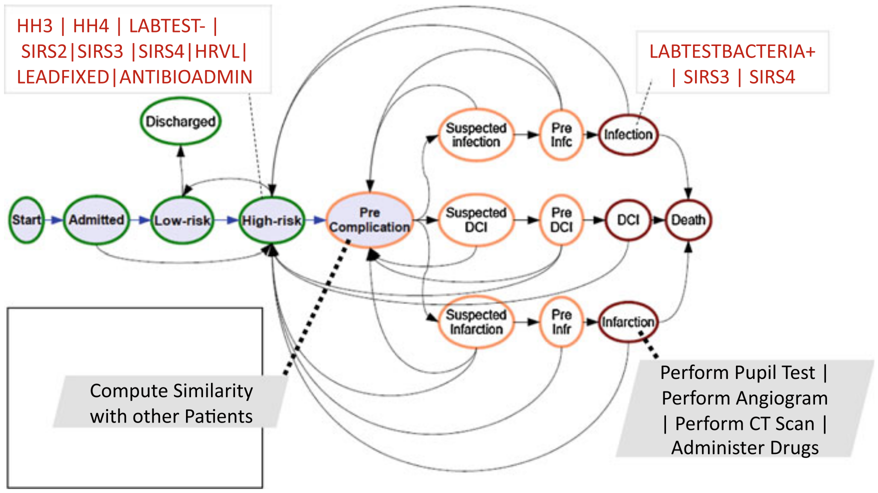


Fig. 10.1 Patient complication detection

diagnosis task. We next describe two real-world models for patient monitoring and cybersecurity analysis derived from experts, and the encoded LTS++ language.

To help describe the patient monitoring application, we describe a state transition model that was drawn by talking with neuro ICU physicians from the Columbia University Neurological unit. Figure 10.1 shows this state transition model. The states have types, *good* drawn with a green outline or *bad*, drawn with an orange or red outline. Each state has a name, and has associated observations that it explains, and actions that it triggers. Note, these actions are analytic actions not to be confused with planning actions. In the figure, the bad states correspond to critical states of a patient such as *Infection*, *DCI*, *Infarction*, or sometimes even a terminal state such as *Death*. The good states are the non-critical states. Upon admission the patient is classified as either in *Lowrisk* or in *Highrisk*. From a *Highrisk* state, they may get to the *Infection*, *Infarction*, or the *DCI* state through intermediate precomplication states. These intermediate states represent states where some clinical signals are present, but before the appearance of definitive symptoms. The patient's condition may improve; hence, the patient's state may move back to the *Lowrisk* state from for example the *Infection* state, based on interventions that the physicians can perform.

Observations in the model are computed from raw data captured by patient monitoring devices (e.g., the patient's blood pressure, heart rate, temperature) as well as other measurements and computations provided by doctors and nurses. In the figure, a subset of all possible observations is shown in light rectangular boxes—attached to the state that explains them. Examples of observations include measures computed from physiological parameters, such as the Systemic Inflammatory Response Syndrome (SIRS) score, may be provided by doctors, such the Hunt and Hess score (HH), and may be the result of performing lab-tests on the patient, such as LabTestBacteria+ (positive test for Bacteria). As is shown, the same observation

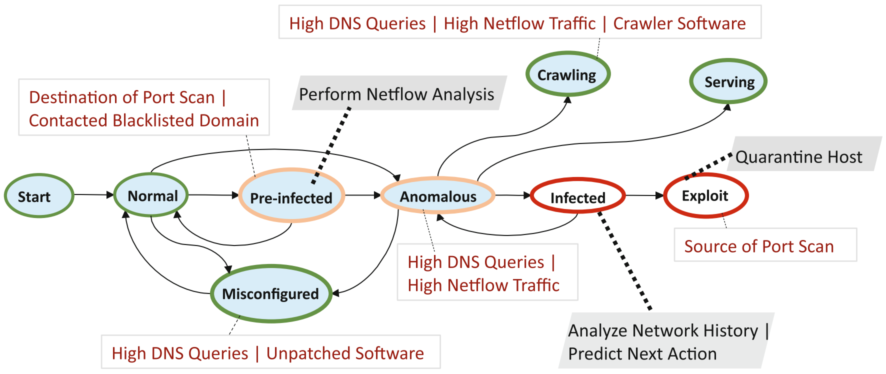

High DNS Queries | High Netflow Traffic | Crawler Software
Crawling
Serving
Destination of Port Scan | Contacted Blacklisted Domain
Perform Netflow Analysis
Quarantine Host
Start
Normal
Pre-infected
Anomalous
Infected
Exploit
Source of Port Scan
High DNS Queries | High Netflow Traffic
Misconfigured
Analyze Network History | Predict Next Action
High DNS Queries | Unpatched Software

Fig. 10.2 Malware detection

can have multiple interpretations. For instance, the patient may have a SIRS score 3 or 4 both in *Highrisk* as well as in *Infection* state. Also shown in the figure—within the shaded parallelograms—are corresponding actions that may be taken when the patient is identified to be in that state. For instance, when the patient is estimated to be in the *Precomplication* state, it is recommended to look at similarity with other patients to further diagnose the patient's condition. Such similarity can be computed using different analytics. Alternately, when the patient is identified as being in a complication state, the actions can correspond to interventions that the physicians need to perform, e.g., *Perform Pupil Test*.

Figure 10.2 shows a state transition model for network host (entity) monitoring in a computer network. This model is derived through consultation with cybersecurity and network monitoring analysts. Bad states in this model correspond to the malware lifecycle, with the host becoming infected with malware. These states include the *Infection* and *Exploit* states shown in the figure. Good states include states associated within the normal modes of operation of a host such as *Crawling*, *Normal*, *Serving* (server behavior), etc. There are also intermediate states that are not completely indicative of infection, but may be pre-cursors to bad states. These include the *Anomalous* and *Pre-Infected* states. Observations for this model are computed from the raw network traffic, including measurements from Domain Name System (DNS) queries, from Netflow measurements, and from Firewall alerts, as well as looking at the network behavior of the hosts. As with the ICU model the same observation of *High Number of DNS Queries* may be associated with either *Anomalous* behavior for end-user machines or *Crawling* behavior. Actions in this case represent analytic tasks, such as *Analyze Network History* to identify how a machine got infected, or physical tasks, such as *Quarantine Host* when the host is identified as being in the *Exploit* state.

While the complexity of the analysis involved to derive observations from the raw data can vary, it is important to note that observations, in both cases described earlier, are by nature *unreliable*:

The Set of Observations Can Be Incomplete Operational constraints will prevent us running in-depth analysis on *all* of the data all the time. However, observations are typically time stamped, and can be temporally ordered.

Observations May Be Ambiguous This is depicted in multiple examples in Figs. 10.1 and 10.2, where the same observation may be explained by many states.

Not All Observations Are Explainable, Given Other Observations There are several reasons while some observations may remain unexplained: (1) observations are (sometimes weak) indicators of a behavior, rather than authoritative measurements; (2) the model description is by necessity incomplete, unless we are able to design a perfect model; (3) in the case of malware detection, malware could try to confuse detectors by either hiding in normal traffic patterns or originating extra traffic.

For instance, consider the cybersecurity model, and a sequence of observations: *Monitoring On*, *High DNS Queries*. This can be explained by the following state sequences:

- *Normal* → *Misconfigured*
- *Normal* → *Anomalous* → *Misconfigured*
- *Normal* → *Misconfigured*
- *Normal* → *Anomalous* → *Crawling*

and other observations are required to disambiguate these states. Some of these may be pre-cursors to infection (e.g., *Anomalous*) and so require careful analysis. Given a sequence of observations and the model, the hypothesis generation task infers a number of plausible hypotheses about the evolution of the entity. In practice this requires a high degree of human skill to perform. By encoding domain knowledge using a simple model of the form described, and coupling with an automated technique that allows for incomplete state transition models, and unreliable observations, we can provide action decision support to human experts. The result of our automated technique is presented as recommendations to physicians or network analysts, or may be used automatically to drive additional analyses.

3 Hypothesis Generation Problem

In this section, we formally define the hypothesis generation problem. To do so, we first define a dynamical system that can model the system behavior. We then define a notion of a hypothesis and hypothesis "plausibility."

A dynamical system is a tuple $\Sigma = (F, A, I)$, where F is a finite set of fluent symbols, A is a set of actions, and $I \subseteq F$ defines the initial state. Actions are defined by their precondition and effects, over the set of fluents F. The set of actions A includes both actions that account for the possible transitions in the model as well as the *discard* actions, one per each observation o with precondition ¬o and no effect. The "*discard*' actions to simulate the "explanation" of an unexplained observation.

That is, the instances of the *discard* action add transitions to the system that account for leaving an observation unexplained. The added transitions by the *discard* action help us define the satisfaction of observations as we will discuss next.

A *system state* s is a set of fluents which defines all that is true in a particular state of a dynamical system. For a state s, let $M_s : F \rightarrow \{true, false\}$ be a truth assignment that assigns $true$ to f if $f \in s$ and $false$ otherwise. An action a is *executable* in a state s if all of its preconditions are met by the state s or $M_s \models c$ for every c in the precondition of a. We define the successor state as $\delta(a, s) = (s\setminus$ delete effects of $a) \cup$ (add effects of action a) for the executable actions. The sequence of actions $[a_1, \dots, a_n]$ is executable in s if the state $s' = \delta(a_n, \delta(a_{n-1}, \dots, \delta(a_1, s)))$ is defined; henceforth, is executable in Σ if it is executable from the initial state.

Let $T \subseteq F$ be the set of fluents that are observable. An *observation* is a fluent in T. Observation formula φ or what we call a *trace* is a sequence of observations. While in general the observation formula φ can be expressed as an Linear Temporal Logic (LTL) formula [4], we consider the trace φ to have the form $\varphi = [o_1, \dots, o_m]$, where $o_i \in T$, with the following standard LTL interpretation[1]:

$$o_1 \wedge \bigcirc\Diamond(o_2 \wedge \bigcirc\Diamond(o_3 \dots (o_{n-1} \wedge \bigcirc\Diamond o_n) \dots))$$

Note that the observations are totally ordered in the above formula. It is typical for the applications we consider to have observations that are timestamps and hence are considered to be totally ordered.

Intuitively, not all observations can be explained; hence, we define the notion of satisfaction of a trace which considers an observation *satisfied* if it is explained or discarded as long as the order of which observations are considered is met by the action sequence. More formally, we define the satisfaction of a trace φ by an action sequence π in Σ as follows.

Definition 1 A trace $\varphi = [o_1, \dots, o_m]$ is satisfied by an action sequence $\pi = [a_1, \dots, a_n]$ if π is executable from the initial state and there is a non-decreasing function f that maps the observation indices $j = 1, \dots, m$ into action indices $i = 1, \dots, n$, such that for all $0 \leq j \leq m$, either $o_j \in s$, where s is the state reached after execution of action $a_{f(j)}$, or $discard_{o_j} = a_{f(j)}$

Consider the following set of actions: A_{o_1} with effect o_1, A_{o_2} with effect o_2, A_{o_3} with effects o_2 and o_3, and action A_{o_4} with effects o_1, o_2, and o_4. Then the trace $[o_1, o_2]$ is satisfied by action sequence $[A_{o_1}, A_{o_2}]$ $(f(1) = 1, f(2) = 2)$, $[A_{o_4}]$ $(f(1) = 1, f(2) = 1)$,$[discard_{o_1}, A_{o_2}]$ $(f(1) = 1, f(2) = 2)$, but not by the action sequence $[A_{o_2}, discard_{o_1}]$ or $[A_{o_1}, discard_{o_1}]$. This is because the order of observation must be met by the function f. No such function would exist for $[A_{o_2}, discard_{o_1}]$. Additionally, an action may explain multiple observation. For example, action A_{o_4} explains both o_1 and o_2; hence, the function f maps both observations to the same action index.

[1] $\bigcirc$ is a symbol for next, $\Diamond$ is a symbol for eventually.

Definition 2 Given the dynamical system description $\Sigma = (F, A, I)$, and a trace $\varphi = [o_1, \ldots, o_m]$, an observation $o_i \in \varphi$ is said to be *ambiguous* if there are at least two actions in A that have the fluent o_i as part of their effects. Further, if φ is satisfied by an action sequence $\pi = [a_1, \ldots a_n]$, an observation o is said to be *missing* from the trace if (1) o is observable (i.e., $o \in T$); (2) $o \notin \varphi$; and (3) o is part of an effect of at least one action a_i in the action sequence π, and $o \in \varphi$ is said to be *noisy* if o is never added by any of the actions $a_i \in \pi$.

According to the above definition, observation o_1 is ambiguous because both action A_{o_1} and action A_{o_4} may explain it. Also given a trace $\varphi = [o_1, o_2]$, φ is satisfied by the action sequence $[A_{o_1}, A_{o_3}]$ and in that case, observation o_3 is said to be missing from the trace φ because o_3 is part of the effect of A_{o_3}, but not in the given trace. Furthermore, o_1 is said to be noisy given the action sequence $[discard_{o_1}, A_{o_2}]$ because o_1 is not added by any of the actions in the plan.

A hypothesis is the sequence of actions that explains the given trace. In the case of unreliable observations, a hypothesis may not explain all the observations by discarding some. Hence, we use our definition of a trace satisfied by an action sequence to formally define a hypothesis as follows.

Definition 3 Hypothesis generation problem is a tuple $HG = (\Sigma = (F, A, I), \varphi)$, where Σ is a dynamical system and φ is the given trace. A hypothesis for HG is a sequence of actions $\pi = [a_1, \ldots, a_n]$, $1 \le i \le n$, $a_i \in A$ such that the trace φ is satisfied by the sequence of actions π.

Given a trace, there are many possible hypotheses, but some could be stated as more plausible than others. Hence, we define a notion of plausibility of a hypothesis. A hypothesis π is said to be at least as plausible as hypothesis π', stated as $\pi \preceq \pi'$, where $\preceq$ is assumed to be a reflexive and transitive plausibility relation.

Definition 4 Given a hypothesis generation problem $HG = (\Sigma = (F, A, I), \varphi)$, π is the most plausible hypothesis for HG if and only if π is a hypothesis for HG and there does not exists another hypothesis π' for HG such that π' is more plausible or $\pi' \preceq \pi$ and $\pi \not\preceq \pi'$.

Next, we define a few cases for the notion of plausibility between hypothesis. A hypothesis π is at least as plausible as hypothesis π', $\pi \preceq \pi'$, if one or more of the following statements hold: π can explain more observations than π', π is a shorter hypothesis, π has minimum number of designated "unlikely" or "bad" actions. The third criteria is similar to the notion of minimum number of "faulty" actions in a diagnostic setting, based on having an optimistic view on what can go wrong.

Back to our example, the hypothesis $[A_{o_4}]$ is more plausible than for example, $[A_{o_1}, A_{o_2}]$ because it is shorter, and the hypothesis $[A_{o_1}, A_{o_2}]$ is more plausible than the hypothesis $[discard_{o_1}, A_{o_3}]$ because it explains both observations. The third criteria is similar to the notion of minimum number of "faulty" actions in a diagnostic setting, based on having an optimistic view on what can go wrong. Note that a hypothesis may be shorter but have more discard actions or more unlikely actions. We address combining the above plausibility relations using numerical cost values of the underlining planning domain. Therefore, plans with smaller costs are more plausible.

4 Model Description in LTS++

In this section, we will describe our proposed language LTS++, derived from LTS (Labeled Transition System) [12] that can be used to define the domain knowledge by a domain expert. As described in the application section, encoding the domain knowledge is itself a challenge specially if the domain expert is not familiar with AI planning. Hence, we also discuss our knowledge engineering effort that can guide the domain expert in describing their knowledge about a particular application. This knowledge is implicitly the same knowledge captured theoretically by the dynamical system. Furthermore, the LTS++ model description together with a trace encodes the hypothesis generation problem we are trying to solve.

Note, LTS++ does not have a full expressive power of PDDL since it encodes state transitions in a simple "next-state" predicate model. A PDDL encoding allows encoding of richer actions with preconditions and effects. Hence, while we can express the LTS++ language into PDDL, we cannot go from a PDDL encoding of the domain to the LTS++ encoding.

We propose a process that further helps the domain experts in creating a model. Figure 10.3 shows our 7-step creation process for an LTS++ model. The arrows are intended to indicate the most typical transitions between steps. This process is meant to help provide guidance to the new users in developing an LTS++ model. While this process is geared towards our applications, we believe that it also provides insight and inspiration into creation of a practical planning problem. Next, we will describe the basic elements in the description of a model in LTS++ following the steps in the model creation process.

1. **Entity**: The domain expert needs to identify the entity which is what the system monitors. This depends on the objective of the hypotheses generator, the available

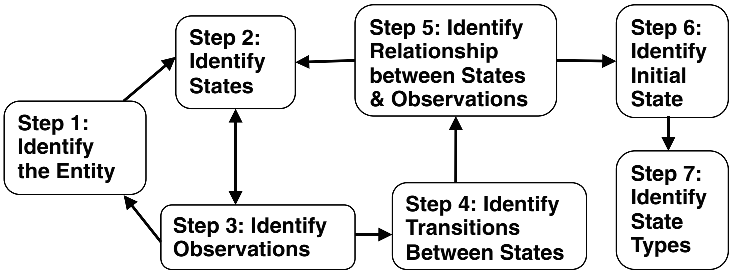
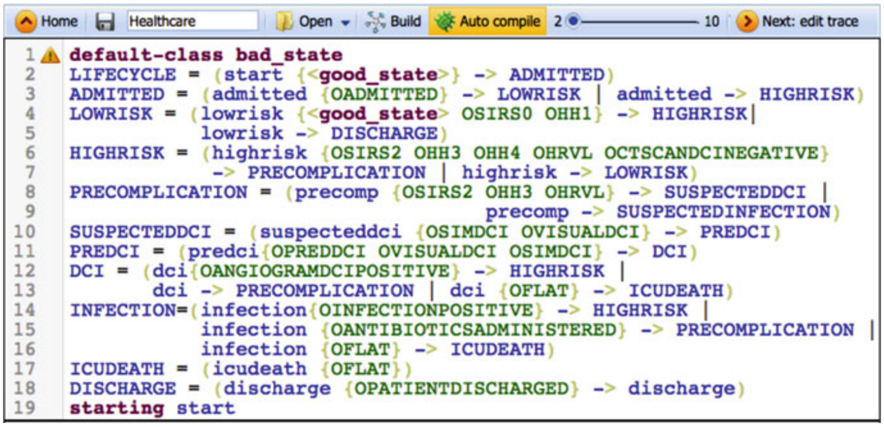

Step 1: Identify the Entity
Step 2: Identify States
Step 3: Identify Observations
Step 4: Identify Transitions Between States
Step 5: Identify Relationship between States & Observations
Step 6: Identify Initial State
Step 7: Identify State Types

Fig. 10.3 Process for LTS++ model creation

data, and the available actions. The entity could be a patient or a host or other objects in the application.

2. **States**: The domain expert needs to identify the possible states of the entity (different from a planning state). States are not directly observable but can be hypothesized. The states of patient for example could be Delayed Cerebral Ischemia (*DCI*), *SuspectedDCI*, *Infection*, *Precomplication* or *Highrisk*. The states could form a hierarchy, in which case all non-child states are called *hyperstates*. For example, there could be multiple precomplication states, each a child of *Precomplication* hyperstate. Designating a state as a hyperstate is useful when it comes to modeling incomplete model and unreliable observations. For example, if a transition through one or several states of the hyperstate is required, but no specific observation is associated with the transition, the hyperstate itself is included as part of the hypothesis, indicating that the model may have a missing state within the hyperstate, and that state in turn may need a new observation type associated with it.
3. **Observations**: The domain experts need to identify a set of observation types that the system needs to reason about. Since observations are received from analytics as a result of analyzing raw data, the available data and analytics may limit the space of observations. Heart Rate Variability Low (*OHRVL*), is an example of an observation. It is important to note that observations are by nature unreliable: the set of observations will be incomplete, observations may be ambiguous, and not all observation will be explainable.
4. **State Transitions**: The domain expert has to describe possible transitions between states. An example transition is going from state *Infection* to *Highrisk*. This transition reflects an improvement in patient state, without describing the cause of this transition. Enumerating all possible transitions may be a tedious task, depending on the number of states. However, one can use hyperstates to help manage these transitions. Any transition into (or out of) a hyperstate is carried to every child of the hyperstate.
5. **Association between States and Observations**: The domain expert has to associate observations to states meaning that this observation *can* be explained by this state. Note, this association can be many-to-many as observations could be ambiguous or indicative of more than one state, and each state can be associated with multiple observations. The observation *OHRVL* is an example of an ambiguous observation because it can be associated with multiple states. Note we add "*O*" to the observation as a convention.
6. **Initial State**: The domain expert can also define the initial state if the initial state is known. For example in the case of healthcare, the initial state can be the state *Admitted*.
7. **State Types**: States could also have types such as, unlikely, or "bad" states. This maps to the notion of "faulty" or "unlikely" planning actions.

Figure 10.4 shows an LTS++ model description for our healthcare application. The states are shown in blue. The observations are specified within the curly brackets and are shown in green. Multiple observations can be separated by



```
default-class bad_state
LIFECYCLE = (start {<good_state>} -> ADMITTED)
ADMITTED = (admitted {OADMITTED} -> LOWRISK | admitted -> HIGHRISK)
LOWRISK = (lowrisk {<good_state> OSIRS0 OHH1} -> HIGHRISK|
           lowrisk -> DISCHARGE)
HIGHRISK = (highrisk {OSIRS2 OHH3 OHH4 OHRVL OCTSCANDCINEGATIVE}
            -> PRECOMPLICATION | highrisk -> LOWRISK)
PRECOMPLICATION = (precomp {OSIRS2 OHH3 OHRVL} -> SUSPECTEDDCI |
                                 precomp -> SUSPECTEDINFECTION)
SUSPECTEDDCI = (suspecteddci {OSIMDCI OVISUALDCI} -> PREDCI)
PREDCI = (predci{OPREDDCI OVISUALDCI OSIMDCI} -> DCI)
DCI = (dci{OANGIOGRAMDCIPOSITIVE} -> HIGHRISK |
       dci -> PRECOMPLICATION | dci {OFLAT} -> ICUDEATH)
INFECTION=(infection{OINFECTIONPOSITIVE} -> HIGHRISK |
           infection {OANTIBIOTICSADMINISTERED} -> PRECOMPLICATION |
           infection {OFLAT} -> ICUDEATH)
ICUDEATH = (icudeath {OFLAT})
DISCHARGE = (discharge {OPATIENTDISCHARGED} -> discharge)
starting start
```

Fig. 10.4 Healthcare model description in LTS++

whitespace or a comma. The state types are specified within angle brackets with a default state type shown in the first line. The transitions between states are specified using arrows. Multiple transitions between states can be specified using a vertical bar. The starting state is specified in the last line.

The knowledge encoded in the LTS++ model is implicitly the same knowledge in the theory of the dynamical system. Informally, each state can be thought of as a label for a subset of planning states of interest and therefore be modeled using a special fluent such as "(at-state)." Each observation belongs to the set T of observable fluents. The state transitions together with the relationship between states and observations define the set of actions A such that the specified state transition is ensured and the observations are part of the effect of the actions. The initial state can also map directly to I. The state types can also map to fluents. Hence, the hypothesis generation problem can now be captures using the LTS++ model description together with a provided trace.

4.1 From LTS++ to a Planning Problem in PDDL

In this section, we describe the planning problem using one fixed encoding of the planning domain, (i.e., description of planning actions, predicates), but varied the planning problem/instance (i.e., initial state, goal state, and variables) based on the given LTS++ model and the given observations. The planning domain is shown in Fig. 10.5 and the planning problem is shown in Fig. 10.6.

The planning domain in Planning Domain Definition Language (PDDL) [13] includes a total of 6 actions. In short, we use two phases, state transitions and explaining or discarding observations and switch between these two using the "ready" predicate. Each transition is followed by either explaining, explain-

```
(:action explain-observation
  :parameters(?x - state ?obs1 - obs ?obs2 - obs ?cat - obs-type)
  :precondition (and (is-next-obs ?obs1 ?obs2)(matches ?obs1 ?cat)
                     (explains ?x ?cat)(at-state ?x)(at-obs ?obs1))
  :effect (and  (not (at-obs ?obs1)) (at-obs ?obs2) (ready)
                     (increase (total-cost) 0)))

 (:action discard-observation
  :parameters(?x - state ?obs1 - obs ?obs2 - obs)
  :precondition (and (is-next-obs ?obs1 ?obs2)(at-state ?x)(at-obs ?obs1))
  :effect (and (not (at-obs ?obs1))(at-obs ?obs2)(ready)
                 (increase (total-cost) 2000)))

 (:action state-change
  :parameters(?x - state ?y - state ?obs - obs)
  :precondition (and (is-next-state ?x ?y)(at-obs ?obs)(at-state ?x)(ready))
  :effect (and (not (at-state ?x))(not (ready))(entering-state ?y)
                   (increase (total-cost) 0)))

 (:action enter-state-good
  :parameters(?y - state ?obs - obs)
  :precondition (and (at-obs ?obs) (entering-state ?y) (good-state ?y))
  :effect (and (at-state ?y)(not (entering-state ?y))
              (increase (total-cost) 1)))

 (:action enter-state-bad
  :parameters(?y - state ?obs - obs)
  :precondition (and (at-obs ?obs)(entering-state ?y)(bad-state ?y))
  :effect (and (at-state ?y)(not (entering-state ?y))
              (increase (total-cost) 10)))

 (:action allow-unobserved
  :parameters(?x - state ?obs - obs)
  :precondition (and (at-obs ?obs)(at-state ?x))
  :effect (and (ready)(increase (total-cost) 1100)))
```

Fig. 10.5 Partial encoding of our sample PDDL domain

```
(:init
  (at-state admitted) (at-obs o_1)(ready)

  (matches o_1 OHH1)(matches o_2 OSIRS0)(matches o_3 OSIRS2)

  (explains lowrisk OSIRS0) (explains highrisk OSIRS2)
  (explains precomp OSIRS2) (explains lowrisk OHH1)
  (explains dci OANGIOGRAMDCIPOSITIVE)
  (explains highrisk OHRVL) (explains precomp OHRVL)

  (is-next-state admitted highrisk) (is-next-state admitted lowrisk)
  (is-next-state lowrisk highrisk) (is-next-state highrisk lowrisk)
  (is-next-state highrisk precomp) (is-next-state dci highrisk)
  (is-next-state dci icudeath) (is-next-state dci precomp)

  (bad-state dci) (bad-state highrisk) (good-state lowrisk)

  (is-next-obs o_1 o_2)(is-next-obs o_2 o_3) (is-next-obs o_3 o_end))

(:goal (and (at-obs o_end) (ready)))
```

Fig. 10.6 Partial encoding of our sample PDDL problem for the intensive care application

observation, or discarding, discard-observation, an observation or moving to the next state transition without explaining or discarding any observation, allow-unobserved which is useful in order to allow missing observations. The explain action has a cost of 0, the discard-observation action has a high cost (e.g., 2000), and the unobserved transition has a cost of 1100 in this encoding. These numbers were set arbitrary here to show the relative comparison between the different costs set for each different action. In practice these numbers can be learned from data but we found that the number we used modeled the behavior as expected.

The predicates "is-next-obs," and "at-obs" are used to keep track of observation order. Observation categories (i.e., ?cat) defines the possible observations in the domain. The predicate "matches" together with the "is-next-obs" defines the current trace or the sequence of observation. The predicate "explains" is used to connect states and observations; "(explains ?x ?cat)" means that state ?x can explain observation of category ?cat. The action explain-observation can explain an observation if the resulting state can explain the observation category of the observation in the trace.

We also had one action, state-change that represents the transitions defined by the actions A. This action had a cost of 0 and the predicate "is-next-state" is used to encode the transitions between states. Two additional actions, enter-state-good and enter-state-bad, are used to associate different costs for good and bad states. The predicates "(bad-state)" and "(good-states)" are used to define the good and bad states in the problem. We used a cost of 1 for good states and a cost of 10 for bad states.

This encoding of the domain allowed us to automatically generate multiple problem sets that include different number of observations as well as different transitions. Partial encoding of our sample PDDL problem is shown in Fig. 10.6. This encoding matches our LTS++ description shown in Fig. 10.4 with the following trace [OHH1, OSIRS0, OSIRS2]. The initial state is a special state 'admitted' with transitions to highrisk and lowrisk states. The goal state is encoded by two predicates "(ready)" and the "(at-obs o_end)" predicate to ensure the last observation is considered. The last observation is only considered if all other observations are considered in the order in which they are given.

Theorem 1 *Let P' be a planning problem constructed as above for a given LTS++ model and a trace φ and HG be the corresponding hypotheses generation problem; HG has only state transition actions in which observations are part of their effects and the discard actions. If π is a plan for P' then there exists a hypotheses π' for HG that can be constructed from π by considering only the state transition actions and the discard actions. On the other hand, if π' is a hypotheses for HG, then there exists a plan π for P' that can be constructed from π' by adding the extra actions explain, enter, and allow-unobserved and by modifying the state transition action.*

Proof If π is a plan for P', therefore it is executable from the initial state and the goal is satisfied; each observation is either explained or discarded and the ordering is preserved. Therefore, there is a non-decreasing function that maps the observation indices into the action indices: if the observation is satisfied it maps to the state-change action and if it is discarded, it maps to the “discard” action. Therefore, if only the state transition and discard actions are kept, then the trace is still satisfied. If the state-change action is modified to include the observation fluent as part of its effect, then this is a hypotheses for HG. On the other hand, if π' is a hypothesis for HG, then we can add the extra actions to π' and modify the state-change action to remove the explicit mention of the observation and the trace would still be satisfied. The result is a plan for π.

Note, the exact PDDL encoding of the planning problem P' determines if for each found plan for P' there would be exactly one corresponding hypotheses or multiple. If we used the encoding shown earlier, then for each plan there could be multiple possible hypotheses because of the positioning of the explain action. It is possible to have a more complex planning domain and force a one-to-one relationship between hypotheses and plans. Nevertheless, the above theorem shows that a hypothesis can be found by translating the hypothesis generation problem into a planning problem and using an AI planner to find a plan. The resulting plan can be turned into a hypotheses by a post-processing step that removes the extra actions from the plan. Furthermore, assuming that the costs of the actions in P' model the plausibility notion correctly, then the lowest-cost plan maps to the most plausible hypotheses. More formally,

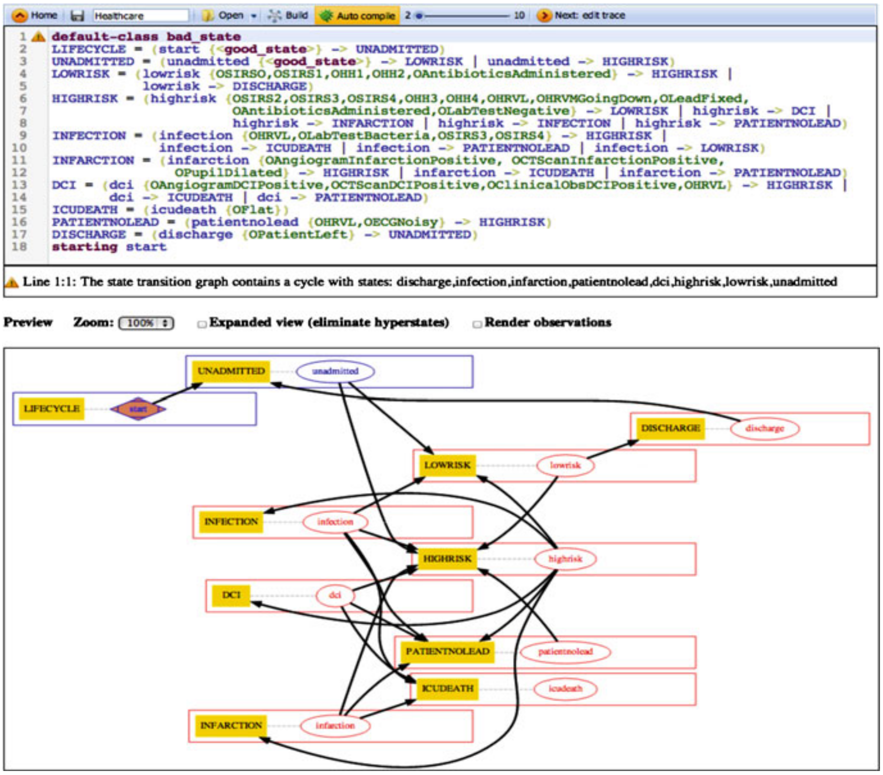
Corollary 1 *Let P' be a planning problem constructed as above for a given LTS++ model and a trace φ and HG be the corresponding hypotheses generation problem. Further, let π_1 and π_2 be two plans for P', and π'_1 and π'_2 be the corresponding hypotheses for HG. Then π'_1 is at least as plausible as π'_2 if and only if $cost(\pi_1) < cost(\pi_2)$.*

Given the association between plans and hypotheses we use top-*k* planning to find a set of plans with low cost. These plans can be translated to hypotheses to find the most plausible hypotheses to the hypotheses generation problem. For details on top-*k* please see [11, 16, 24].

5 LTS++ Integrated Development Environment

LTS++ Integrated Development Environment (IDE) is a web-based tool that helps the domain experts to create model descriptions by describing LTS++ models and to generate hypotheses. LTS++ IDE consists of an LTS++ editor, graphical view of the transition system, specification of the trace, and generation of hypotheses. The tool automatically generates planning problems from the LTS++ specification and the entered trace. The generated hypotheses are the result of running our planner and presenting the result from top-most plausible hypothesis to the least plausible hypothesis.

Model Editor The top part of the model editor screen (Fig. 10.7) is the LTS++ language editor with syntax highlighting and the bottom part is the automatically



```
default-class bad_state
LIFECYCLE = (start {<good_state>} -> UNADMITTED)
UNADMITTED = (unadmitted {<good_state>} -> LOWRISK | unadmitted -> HIGHRISK)
LOWRISK = (lowrisk {OSIRSO,OSIRS1,OHH1,OHH2,OAntibioticsAdministered} -> HIGHRISK |
           lowrisk -> DISCHARGE)
HIGHRISK = (highrisk {OSIRS2,OSIRS3,OSIRS4,OHH3,OHH4,OHRVL,OHRVMGoingDown,OLeadFixed,
                      OAntibioticsAdministered,OLabTestNegative} -> LOWRISK | highrisk -> DCI |
                      highrisk -> INFARCTION | highrisk -> INFECTION | highrisk -> PATIENTNOLEAD)
INFECTION = (infection {OHRVL,OLabTestBacteria,OSIRS3,OSIRS4} -> HIGHRISK |
             infection -> ICUDEATH | infection -> PATIENTNOLEAD | infection -> LOWRISK)
INFARCTION = (infarction {OAngiogramInfarctionPositive, OCTScanInfarctionPositive,
              OPupilDilated} -> HIGHRISK | infarction -> ICUDEATH | infarction -> PATIENTNOLEAD)
DCI = (dci {OAngiogramDCIPositive,OCTScanDCIPositive,OClinicalObsDCIPositive,OHRVL} -> HIGHRISK |
       dci -> ICUDEATH | dci -> PATIENTNOLEAD)
ICUDEATH = (icudeath {OFlat})
PATIENTNOLEAD = (patientnolead {OHRVL,OECGNoisy} -> HIGHRISK)
DISCHARGE = (discharge {OPatientLeft} -> UNADMITTED)
starting start
```

```
Line 1:1: The state transition graph contains a cycle with states: discharge,infection,infarction,patientnolead,dci,highrisk,lowrisk,unadmitted
```

Fig. 10.7 LTS++ IDE

generated transition graph. In the editor, the states appear in blue. The observations are specified within the curly brackets and appear in green. You can specify multiple observations by using space or comma between observations. The transitions between states are specified using arrows. Multiple transitions between states can be specified using a vertical bar. The LTS++ model editor automatically detects errors in LTS++ language and shows them below the text editor.

Model Testing To test the model, a sequence of observations can be entered by clicking on "Next: edit trace" from the LTS++ IDE main page. The tool automatically generates planning problems from the LTS++ specification and entered trace. The generated hypotheses are the result of running a planner and finding the most plausible hypotheses ranked by plausibility from highest to lowest. Figure 10.8 shows an example of hypotheses generated for the critical care model; the result is automatically generated by our tool. Each hypothesis is shown as a sequence of states matched to an observed event sequence. The observations that are explained by a state are shown in green ovals, and unexplained observations are shown in purple. The arrows between the observations show the sequence of

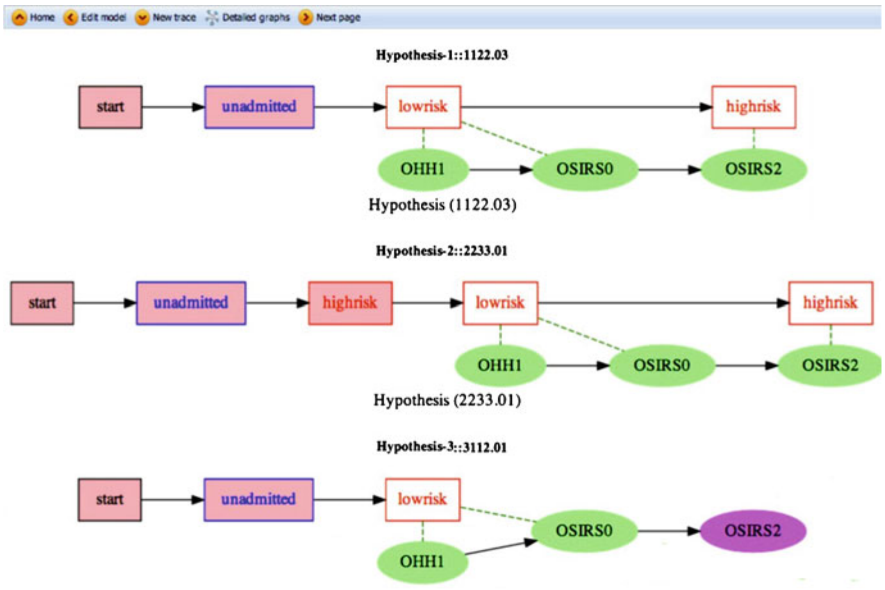


Fig. 10.8 Sample healthcare example

observations in the trace. Each hypothesis is associated with a cost. The lower the cost value, the more plausible is the hypothesis.

Model Discovery and Update Our tool uses a simple bootstrapping technique to discover an initial model given a set of historical observations. Several candidate models will be presented to the domain expert who can choose one as an initial LTS++ model and further improve it. We also implement automated model updates to produce better quality hypotheses as we do not assume the model will be accurate in perpetuity. To do this, we use an aggregate measure of the plausibility of top-N hypotheses as our optimization criteria. Using a genetic algorithm, we attempt small atomic changes to the model (e.g., addition and deletion of states, observations and transitions) and measure the increase in aggregate hypothesis plausibility as a result. In subsequent generations, we combine the promising atomic changes and repeat until we can no longer increase hypothesis plausibility.

Model Composition A single LTS++ model describes a state transition system for a single type of entity, such as a patient. Given multiple entities, each with their own associated model, our tool also allows automated composition of multiple models. It does so by considering a cross product of all possible joint states while paying special attention to the association between observations and the combined states.

Hypothesis Clustering Many of the generated hypotheses are only slightly different from each other. That is, they do seem to be duplicates of each other, except for one or more states or actions that are different. To consolidate similar plans produced by the planner, we apply a clustering algorithm to cluster similar plans and present clusters of plans, where each cluster can be replaced by its representative plan.

6 Related Work

There are several approaches in the diagnosis literature related to the hypothesis generation problem in which use of planners as well as SAT solvers are explored (e.g., [2, 6, 7, 18]). The hypothesis generation problem is also related to the plan recognition problem and the use of AI planning have been explored in that space as well [14, 15, 22]. In particular, Sohrabi et al., explored the same ideas as discussed here with respect to the notion of unreliable observations for the several related problem such as future state projection problem [23] and enterprise risk management [20, 21, 25, 26] that have a corresponding a plan recognition problem. It is important to note that these papers also discuss and address the knowledge engineering challenge through what is called a Mind Map. A Mind Map is a graphical representation of the concepts and relations. The domain knowledge can be encoded by one or more Mind Maps connected by the same concept used in multiple Mind Maps. The Mind Maps can be created in a tool such as FreeMind that produces an XML representation of the Mind Maps and be provided to a system. The system then translates the Mind Maps into an AI planning problem automatically. It is also possible to learn the causal relation between the concepts in order to build the Mind Maps automatically from scratch or augment or validate existing ones [8]. Then similarly, a set of top-*k* or top-quality plans are found through top-*k* planning [11, 16, 24]. Diverse planning [9] or top-quality [10] planning can also be explored to compute such a set of plans.

7 Summary

We presented LTS++, an interactive development environment for planning-based hypothesis generation. To enable our planning-based reasoning, we proposed a characterization of the hypothesis generation problem and showed its correspondence to an AI planning problem. To address the knowledge engineering challenge, we have developed a language, also called LTS++ that allows the domain expert to specify the state transition model and encoding of the observations without any knowledge of AI planning or existing planning languages. LTS++ IDE facilitates model testing and debugging, generating, and visualizing multiple hypotheses for user-provided observations. The tool automatically generates planning problems from the LTS++ specification and the entered trace. The generated hypotheses are the result of running our planner that computes a set of high-quality plans. The hypotheses can be visualized and shown to the analyst or can be further investigated automatically.

References

1. Aljazzar, H., Leue, S.: K*: A heuristic search algorithm for finding the *k* shortest paths. Artificial Intelligence **175**(18), 2129–2154 (2011)
2. Bauer, A., Botea, A., Grastien, A., Haslum, P., Rintanen, J.: Alarm processing with model-based diagnosis of discrete event systems. In: Proceedings of the 22nd International Workshop on Principles of Diagnosis (DX). pp. 52–59 (2011)
3. Cassandras, C., Lafortune, S.: Introduction to discrete event systems. Kluwer Academic Publishers (1999)
4. Emerson, E.A.: Temporal and modal logic. Handbook of theoretical computer science: formal models and semantics **B**, 995–1072 (1990)
5. Göbelbecker, M., Keller, T., Eyerich, P., Brenner, M., Nebel, B.: Coming up with good excuses: What to do when no plan can be found. In: Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS). pp. 81–88 (2010)
6. Grastien, A., Anbulagan, Rintanen, J., Kelareva, E.: Diagnosis of discrete-event systems using satisfiability algorithms. In: Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI). pp. 305–310 (2007)
7. Haslum, P., Grastien, A.: Diagnosis as planning: Two case studies. In: International Scheduling and Planning Applications Workshop (SPARK). pp. 27–44 (2011)
8. Hassanzadeh, O., Bhattacharjya, D., Feblowitz, M., Srinivas, K., Perrone, M., Sohrabi, S., Katz, M.: Answering binary causal questions through large-scale text mining: An evaluation using cause-effect pairs from human experts. In: Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI) (2019)
9. Katz, M., Sohrabi, S.: Reshaping diverse planning. In: Proceedings of the 34th Conference on Artificial Intelligence (AAAI-20) (2020)
10. Katz, M., Sohrabi, S., Udrea, O.: Top-quality planning: finding practically useful sets of best plans. In: Proceedings of the 34th Conference on Artificial Intelligence (AAAI-20) (2020)
11. Katz, M., Sohrabi, S., Udrea, O., Winterer, D.: A novel iterative approach to Top-k planning. In: Proceedings of the 28th International Conference on Automated Planning and Scheduling (2018)
12. Magee, J., Kramer, J.: Concurrency - state models and Java programs (2. ed.). Wiley (2006)
13. McDermott, D.V.: PDDL—The Planning Domain Definition Language. Tech. Rep. TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control (1998)
14. Ramírez, M., Geffner, H.: Plan recognition as planning. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI). pp. 1778–1783 (2009)
15. Ramírez, M., Geffner, H.: Probabilistic plan recognition using off-the-shelf classical planners. In: Proceedings of the 24th National Conference on Artificial Intelligence (AAAI). pp. 1121–1126 (2010)
16. Riabov, A., Sohrabi, S., Udrea, O.: New algorithms for the top-k planning problem. In: Proceedings of the Scheduling and Planning Applications Workshop (SPARK) at the 24th International Conference on Automated Planning and Scheduling (ICAPS). pp. 10–16 (2014)
17. Riabov, A.V., Sohrabi, S., Sow, D.M., Turaga, D.S., Udrea, O., Vu, L.H.: Planning-based reasoning for automated large-scale data analysis. In: Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS). pp. 282–290 (2015)
18. Sohrabi, S., Baier, J., McIlraith, S.: Diagnosis as planning revisited. In: Proceedings of the 12th International Conference on the Principles of Knowledge Representation and Reasoning (KR). pp. 26–36 (2010)
19. Sohrabi, S., Baier, J.A., McIlraith, S.A.: Preferred explanations: Theory and generation via planning. In: Proceedings of the 25th National Conference on Artificial Intelligence (AAAI). pp. 261–267 (2011)
20. Sohrabi, S., Katz, M., Hassanzadeh, O., Udrea, O., Feblowitz, M.D.: IBM scenario planning advisor: Plan recognition as AI planning in practice. In: Proceedings of Demonstration Track at the 27th International Joint Conference on Artificial Intelligence (IJCAI-18) (2018)

21. Sohrabi, S., Katz, M., Hassanzadeh, O., Udrea, O., Feblowitz, M.D., Riabov, A.: IBM scenario planning advisor: Plan recognition as AI planning in practice. AI Commun. **32**(1), 1–13 (2019)
22. Sohrabi, S., Riabov, A., Udrea, O.: Plan recognition as planning revisited. In: Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI). pp. 3258–3264 (2016)
23. Sohrabi, S., Riabov, A., Udrea, O.: State projection via AI planning. In: Proceedings of the 31st Conference on Artificial Intelligence (AAAI-17). pp. 4611–4617 (2017)
24. Sohrabi, S., Riabov, A., Udrea, O., Hassanzadeh, O.: Finding diverse high-quality plans for hypothesis generation. In: Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI). pp. 1581–1582 (2016)
25. Sohrabi, S., Riabov, A., Udrea, O., Yuan, F.: Using lightweight semantic models to assist risk management in a large enterprise. In: Proceedings of the 16th International Semantic Web Conference - Industry Track (ISWC-17) (2017)
26. Sohrabi, S., Riabov, A.V., Katz, M., Udrea, O.: An AI planning solution to scenario generation for enterprise risk management. In: Proceedings of the 32nd National Conference on Artificial Intelligence (AAAI). pp. 160–167 (2018)
27. Sohrabi, S., Udrea, O., Riabov, A.: Hypothesis exploration for malware detection using planning. In: Proceedings of the 27th National Conference on Artificial Intelligence (AAAI). pp. 883–889 (2013)