Execution Knowledge for Execution Monitoring: what, why, where and what for?

Štefan Konečný* and Federico Pecora* and Alessandro Saffiotti*

***Abstract*— Despite the progress made in planning and robotics, autonomous plan execution on a robot remains challenging. One of the problems is that (classical) planners use abstract models which are disconnected from the sensor and actuation information available during execution. This connection is typically created in a non-systematic way by some system-specific execution software. In this paper we propose to explicitly represent Execution Knowledge that encodes the connection between planning models and the actual actions and observations for a given physical system. We present an execution monitoring framework in which Execution Knowledge captures the expectations about physical plan execution. A violation of these expectations indicates an execution failure.**

I. INTRODUCTION

In the infancy of AI and Robotics the problems of planning and plan execution monitoring were closely intertwined. Automated planning was meant to animate robots, and robots were meant to execute automatically generated plans. This paradigm was embodied in Shakey the Robot which was running STRIPS [1] to plan and PLANEX [2] to execute plans. Later, planning and robotics went their separates ways addressing problems prevalent in their respective domains: abstract planning, and executing actions on robots. Because of this, the question: "How to autonomously execute and monitor the execution of automatically generated plans?" has been under-addressed.

Classical planners rely on *Planning Knowledge*, an abstract model describing the properties and the dynamics of the system. When the plan is executed, the real evolution of the system may differ from the one anticipated during planning. This is due to action failures, noisy sensors and actuators, and environment dynamics independent of the robot which were abstracted away in the *Planning Knowledge*. The robot can partially observe the complete state of the system through its sensors and information from its actuators. In this paper we call this information *Situational Knowledge*.

Execution Monitoring typically consists of three subtasks: fault detection, fault identification and fault recovery. This paper focuses on fault detection, which is the process of assessing whether there is a significant discrepancy between the observed state of the system and the state anticipated at planning time. Fault identification deals with identification of different faults and fault recovery specifies how to recover from faults.

Planning Knowledge captures the system at the level of detail necessary for means-end reasoning and it typically assumes successful execution of planned actions. It does not describe how this assumption can be verified or contradicted from the available Situational Knowledge for a given physical system. In this paper we focus on this additional information, which we call *Execution Knowledge*, representing the expected future observations resulting from a nominal (or failed) execution. We encapsulate all platform and domain-dependent information about physical execution into Execution Knowledge.

A good reason to keep Execution Knowledge separate from Planing Knowledge is the possibility to employ different representation and reasoning mechanisms. We will discuss how qualitative temporal relations — not included in Planning Knowledge — can be used for fault detection in a scenario involving a PR2 robot[1] that delivers a mug.

Execution Knowledge is not reduced to describing the nominal execution or expected failures of an individual action w.r.t. to Situational Knowledge. It may explicitly describe relations between observations related to different actions. The dependency between these actions may be specific to the executed plan and to the physical robotic system. Expressing these relations in the Planning Knowledge is often cumbersome from the representational and/or computational point of view.

In this paper we propose an Execution Monitoring framework in which separate Planing and Execution Knowledge are compiled into a constraint based representation of past observations, expected future observations, and relations between all these — called the *Execution Network*. This Execution Network is updated by an Execution Monitor which decides whether a received observation is relevant to the current execution (as specified by the Execution Network). The Execution Monitor also generates complex observations from simpler ones, detects failures, and dispatches actions based on information from the Execution Network.

II. THE EXECUTION MONITORING FRAMEWORK

Figure 1 shows how the separate sources of knowledge – Planning Knowledge (PK), Execution Knowledge (EK) and Situational Knowledge (SK) – are used for execution monitoring. PK is used by the Planner to generate a plan. EK contains additional expectations about execution and specifies how to validate these expectations based on available SK for the specific environment and platform. The Network Constructor takes PK, the plan, and EK to create

* Center for Applied Autonomous Sensor Systems, Örebro University, S-70182 Örebro, Sweden. Email: {sky, fpa, asaffio}@aass.oru.se

[1] PR2 (Personal Robot 2) is a robotic platform developed by Willow Garage http://www.willowgarage.com

the Execution Network (EN). The EN represents instantiated expectations about the execution of a plan. Nodes in EN represent the expected actions and observations, while edges represent the expected qualitative temporal relations between the nodes.

The EN is maintained by the EM during execution. For each observation, the EM checks whether it is relevant to the execution, i.e., whether it corresponds to a node in the EN (an expectation). If so, the observation is added to the EN, and a link (called an "anchor") is established between the two nodes (the observation and the expectation). If an anchored observation is updated so that it violates some expectation, the EM detects a failure. The EM also dispatches actions based on temporal information encoded in EN.

The practical difficulties connected to plan execution, and therefore the interest to distinguish and separate the above three types of knowledge, can be appreciated already from very simple examples. Below we use the well known Blocks World domain, where a robotic arm is employed to stack blocks on a table.

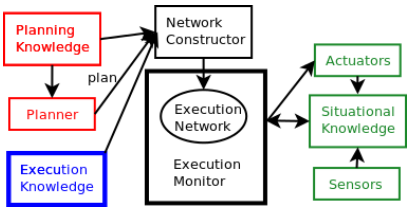

Fig. 1. Execution Monitoring framework. The modules are coloured according to the knowledge they use. The black modules use multiple forms of knowledge. This paper focuses on Execution Knowledge and the Execution Monitor that maintains the Execution Network, which are marked by a thick.

A. The Blocks World Scenario

The Blocks World (BW) is a prototypical example of a planning problem in which many abstractions are made: the robotic arm is either holding a block or it is empty; it can move only in two dimensions — up until the maximum height, down until it reaches the topmost block of a stack, and in discrete steps left and right to build stacks. The goal is to achieve a desired configuration of blocks[2].

B. Planning Knowledge

The level of abstraction of the considered actions can range from low level commands sent to the crane's servos to high level abstract instructions. Using the latter approach is more practical for several reasons: the resulting plan is comprehensible to humans, as it abstracts away the physical properties of the crane and of the environment, such as whether the connection between the rail and gripper is soft or rigid, or the weight and size of the blocks; an abstract domain allows to use techniques developed in classical planning, and to reuse the same planning representation across different (constructions of) cranes and environments.

[2]Even under these extremely simplifying assumptions, most interesting problems in the Blocks World domain are NP-Hard [3].

For the sake of the present discussion, we adopt a STRIPS-like representation of the planning domain. The World Model represents the abstract state of the world by a set of logical predicates and the closed world assumption is adopted. It also specifies the circumstances in which an action can be executed through preconditions and how a *successful* execution of an action will alter the world state through effects. All this knowledge constitutes PK.

In the Block Worlds domain `unstack(?b1,?b2)` can be executed only if the crane is above the concerned block (`above(?b1)`), the block is on top of the stack (`top(?b1)`), and it is supported by some other block `?b2` (`on(?b1,?b2)`). These are the preconditions of the action. If the action `unstack(?b1,?b2)` is executed successfully in the physical environment, the gripper will hold `?b1` and `?b2` will be the top block. To reflect this the PK has to specify that `holding(?b1)`, `top(?b2)` should be added and `on(b1)`, `top(b2)` should be removed from the PK.

`moveOver(?l1,?l2)` has one precondition, namely `above(?l1)`. It has two effects: the old location `above(?l1)` is removed from PK and the new location `above(?l2)` is added into PK. We shall focus our discussion on these two actions. Our analysis can be easily adapted for other actions used in this domain: `stack`, `pick`, and `place`.

C. Situational Knowledge

SK is the knowledge the robot has about the current state of the system. It consists of *directly* perceived information from robot's actuators and sensors; and information *derived* from it, which will be discussed in the EK section.

To reason about the temporal aspects of SK within EK, SK has to represent time. All observations in SK and expectations in EK are expressed through fluents:

Definition 1: A *fluent* is a pair $\psi = (p, [st, ft])$, where

- p is a ground atom $p = P(t_1, \ldots, t_n)$ with terms $\{t_1, \ldots, t_n\}$, where P is a predicate symbol
- $[st, ft]$ is an interval where $st, ft \in \mathbb{N}$ represent the start time and finish time of the fluent, respectively.

For simplicity we assume discrete time. p can represent information directly available from sensors (direct observations) or elements from PK which have to be verified during Execution Monitoring (derived observation).

In our BW scenario, the crane has controllers for vertical and horizontal motion, and for the gripper. These controllers communicate their state, which contributes to SK. The vertical controller cY (changing the y coordinate of the gripper) can be in one of the three states $\{lower, raise, idle\}$. $\psi = (cY(lower), [5, \infty))$ represents that the vertical controller started to lower the gripper at time 5 and the movement is still being executed. $\psi = (cY(lower), [5, 10])$ specifies that the controller has stopped the lowering movement at time 10. For brevity we shall refer to a fluent by its predicate, i.e., $cY(lower)$ instead of $\psi = (cY(lower), [5, 10])$ for the rest of this paper. Note that we do not know whether the movement has actually occurred in the physical world, we only know that the controller was active. The horizontal con-

troller cX can be in states $\{left, right, idle\}$. The gripper controller cG can be in states $\{open, close, idle\}$.

Proprioception produce (pX, pY) representing the exact continuous x and y coordinates. pH is a discretized version of pY w.r.t. to the block size — $pH(3)$ indicates that gripper is at the height to grasp the top block from a stack of 3. pG is produced by a pressure sensor inside the gripper and has the two possible values $\{open, close\}$.

A camera placed inside the gripper can observe the top block, e.g., $oa(b1)$ for $b1$, or detect $oa(t1)$ if there is no stack. The camera sight is occluded when a block is held, resulting in $oa(occluded)$. The known size of the block can be used to estimate the discretized distance od (w.r.t. block size) between the camera (and the gripper) and the block.

The fluent $holding$ is produced by the EM based on EK which, we will discuss next.

D. Execution Knowledge

EK contains generic knowledge about the physical execution of actions given the specific platform and environment. For instance, EK specifies the expected temporal relations between the state of the controllers and the position of the crane. It may also include geometric parameters like the size of the blocks or their precise position. EK is specific to a given scenario, environment and physical platform.

In this paper we use Allen Interval Algebra [4] (AIA) relations to capture temporal expectations about execution. These binary relations impose restrictions on temporal ordering of start times and finish times of the two involved fluents (see Figure 2). We use AIA relations because they conveniently represent flexible temporal relations and allow for temporal reasoning in polynomial time [18], [17]. EK contains relations between variables (e.g., $?b1$). Relations between instances (e.g., $b1$) are generated by the Network Constructor.

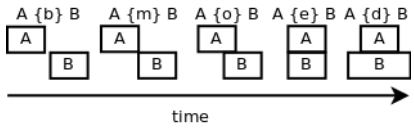

Fig. 2. Graphical representation of some of the Allen Interval Algebra relations: before (b), meets (m), overlaps (o), equals (e) and during (d).

For `moveOver(?l1,?l2)`, EK encodes specific temporal expectations about the the preconditions and effect of the action. The expectation that the gripper has to be $above(?l1)$ before cX is activated can be expressed execution through $oa(?b1)\{o\}cX(left)$. Since the exact position of `?b1`, $posX(?b1)$, is specified in EK, we could use proprioception — $pX(posX(?b1))$ instead of $oa(?b1)$. EK may also encode information about how observations relate to one another. For example, the fact that proprioception and perception should not contradict each other is expressed through $pX(posX(?b1)\{e\}oa(?b1)$. Violation of this relation indicates that at last one of the sensors is unreliable.

If the gripper is holding $?b1$, $b1$ could be dropped during the movement. Since PK assumes that every action will succeed, this is not modelled in PK. We can use proprioception to detect the opening of the gripper during movement: $cX(left)\{d\}pG(closed)$. With $cX(left)\{d\}oa(occluded)$ perception can be used to the same end.

Execution of the `unstack(?b1, ?b2)` action requires three movements: lowering the open griper, closing it, and raising it. This translates in the expected controller activations $cY(lower)_1\{m\}cG(close)$ $\{m\}cY(raise)$. The lowering must reach the appropriate height $h = posY(?b1)$. EK can be designed so that a failure in the executed behaviour signals a specific error. The gripper can be lowered to a slightly lower height $h = posY(?b1) - \epsilon$ so as to "tap" the top of the stack and see whether it is not lower than expected. If that were the case, $cY(lower)\{d\}pH(posY(?b1))$ would be observed, instead of the expected $cY(lower)\{o\}pH(posY(?b1))$. For a stack taller then expected, the observed relation would be $\{b\}$.

When the stack has the expected height, the top block can be grasped, i.e., $cG(close)\{d\}pH(posY(?b1))$. If $pH(posY(?b1))\{o\}pG(closed)$ is indeed detected during execution, the derived fluent $holding(b1)$ is added into SK by the EM, and its start time should be set to the time of observation. This case indicates that EK can be used not only to detect failures, but also to inferred "higher-level" knowledge (in this case, the knowledge of $holding(b1)$). In this example, the added fluent models the occurrence of a predicate which appears in PK, and which is used as the a precondition of stacking and effect of unstacking actions.

A similar set of relations describes the situation when $holding(b1)$ should be closed by the Execution Monitor, i.e., the gripper should be at a proper height to release $b1$. If $pG(open)$ is observed outside of this context, $b1$ was dropped and a failure should be detected. Instead of trying to specify a context when $pG(open)$ cannot occur (e.g., $cX(left)\{d\}pG(closed)$), we try to specify the context when it can, which is much easier.

III. REALIZATION ON A PR2

The BW scenario is very stringent, e.g., it assumes that manipulated object have the same shape, weight and size. The arm is specifically designed to manipulate this specific type of blocks. Yet even so, we have shown how non-trivial EK can be used to capture sophisticated nominal conditions of execution, and to recognize failures that would otherwise be difficult to ascertain through direct observation. We now turn our attention to a sophisticated general purpose platform, a Willow Garage PR2 (see Figure 4). The PR2 has an omnidirectional base, a torso with adjustable height, a head equipped with a stereo camera and a Kinect sensor, and two arms, each equipped with a gripper. We consider an example in which a PR2 has to perform the following task:

> *The robot has to deliver mug* `m1` *to table* `t1`*. It does not hold* `m1`*, but it knows that it stands on counter* `c1`*. It has to execute the following plan (using its right arm* `r1`*):* `drive(c1), dock(c1), observe(c1),`

`pick(m1,r1), undock(), drive(t1), dock(t1), observe(t1), place(m1,r1)`

A. Planning Knowledge

The planning domain has the same level of abstraction as BW. Preconditions and effects in the produced plan are `robotAt(c1)`, `docked(c1)`, `on(m1,c1)`, `holding(m1,c1)`. The corresponding fluents are produced by the EM based on EK and SK.

`drive` and the two object manipulation actions (`pick`, `place`) require platform specific preparation. This preparation is modelled in PK through the action `dock`, an abstraction of a series of specific operations that include adjusting the torso and arm postures so as to be at the same level of the table and prepared for manipulating objects on it (similarly for the action `undock`). Since these auxiliary actions are platform-specific and absent from PK, they will be introduced and discussed in the context of EK. The PR2 can manipulate different objects in different environments. The required motion cannot be pre-specified through a simple schema like in the Blocks World, and is calculated online for the exact position and shape of the manipulated object. A dedicated `observe` action is provided to symbolize the acquisition of these data for `pick(m1,r1)`.

B. Situational and Execution Knowledge

Some fluents in SK are created directly by sensors detecting the state of single actuators (i.e., $pG(gHalfClosed, r1)$ for the gripper on the right arm). Fluents representing the joint state of multiple actuators (i.e., the arm posture $aGrasp$) or more abstract concepts are created from such basic fluents.

EK specifies the exact metric position of the furniture (e.g., $pos(c1)$ for `c1`). It also specifies metric relations describing when `robotAt(c1)` holds given the current position of the robot and of $pos(c1)$. Now that we know how to verify instances of `robotAt`, EK can refer to the general $robotAt(?l1)$ to express expectations about execution. For instance, $drive(?l1)\{o\}robotAt(?l1)$ represents the expectation that `drive(?l1)` will reach its goal.

The auxiliary actions used in EK are $mTorso, mBasemArm$ and $mGripper$. $mTorso$ changes the torso posture, $mArm$ the arm posture. $mBase$ moves the robot very close to the furniture (even sliding the base underneath the table), switching off the obstacle avoidance (which is not the case for $drive$). The exact location where to move is specified in EK and is calculated w.r.t. $pos(c1)$. $mGripper(open, ?a1)$ opens and $mGripper(close, ?a1)$ closes the gripper. The gripper sensor produces fluent $gOpened$, $gClosed$, $gHalfClosed$. The last one is produced if $mGripper(close)$ cannot close gripper completely (due to a blocking object).

After the `dock(?l1)` action, the torso should be high (tUp) to observe and reach $?l1$, and the arm should be in an appropriate posture to manipulate objects on $?l1$ ($aGrasp$). This must be achieved before $mBase$ moves close enough to $?l1$, otherwise $?l1$ could obstruct $mTorso$ and $mArm$. In EK these expectations are expressed through $mTorso(tUp)\{b\}mBase(?l1)$ and $mArm(?a1, aGrasp)\{b\}mBase(?l1)$. Note that EK does not specify the relative order of $mTorso(tUp)$ and $mArm(?a1, aGrasp)$; both actions could be executed in parallel, or a relation forcing the desired ordering could be added into EK. If all the auxiliary actions terminate with success without violating the specified temporal relations, the EM adds a fluent $docked(?l1)$ to SK. Thus, EK specifies how to bridge the gap between PK and directly observed SK.

A similar set of relations is used to describe `undock()`, which should bring the torso low and the arm close to the body to increase stability during `drive`, ideally using the configuration $tDown, aTucked$. To which extent this is possible depends on the size and shape of the manipulated object, i.e., for the `m1` only $tMiddle, aCarry$ is possible. The Network Constructor decides which configuration should be used when it generates the instantiated EN, based on the plan. Again, if all auxiliary actions terminate successfully within expectations then the EM closes the $docked(?l1)$ fluent.

EK specifies that $holding(?o1, ?a1)$ is produced if the gripper tries to close in the presumed location of the object `?o1` and it cannot close entirely due to the grasped object, i.e., $mGripper(close, ?a1)\{o\}pG(gHalfClosed, ?a1)$. This is expected in the context of $mGripper(close, ?a1)\{d\}$ $pick(?o1, ?a1)\{o\}holding(?o1, ?a1)$, $pG(gHalfClosed, ?a1)\{e\}holding(?o1, ?a1)$. When the EM starts the $holding(?o1, ?a1)$ fluent, it also finishes the fluent $on(?o1, ?f1)$, where $?f1$ represents the supporting furniture. A detection of $pG(gHalfClosed, ?a1)$ outside of this context signals a failure. The situation is described in

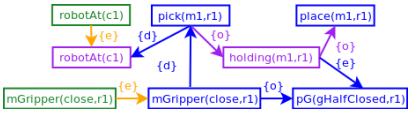

Fig. 3. The state of the EN when the robot has reached `c1` and is closing the gripper.

Figure 3, which represents expectations about $pick(m1, r1)$ at the point in time when the robot has reached `c1` and is closing the gripper. Green boxes represent observed fluents (either through proprioception or perception). Blue and purple boxes represent expectations: blue ones will eventually be anchored to observations, whereas purple ones will eventually be anchored to inferences from observations (e.g., $holding$ can be inferred from gripper status and position). Edges are temporal relations expressed in AIA. The blue ones are modelled in EK (and are therefore unknown to the planner), whereas purple ones are temporal interpretations of causal dependencies in PK (e.g., that $holding$ is not just a consequence of picking, but that it begins to occur before the picking action is over). Yellow edges anchor observations to expectations, and are hence constraints imposing temporal equality.

Analogously, $place(?o1, ?a1)\{o\}holding(?o1, ?a1)$

specifies the context when $holding(?o1, ?a1)$ and $pG(gHalfClosed, ?a1)$ can finish. If $pG(gHalfClosed, ?a1)$ is detected outside of this context, we detect the dropping of `?o1` independently of the executed action (e.g., `undock`, `drive`, `dock`). Which temporal constraints are suitable to capture (PK) relations between preconditions, actions and effects is discussed in [5].

EK can capture relations which are not explicit in the PK. For instance the fact that robot's base stays still during `observe` and `pick` is expressed as $observe(?l1)\{d\}robotAt(?l1)$, $pick(?o1, ?a1)\{d\}robotAt(?l1)$ (see Figure 4). Since the $pick(?o1, ?a1)$ movement is very precise, it should be not attempted if the base moved unexpectedly.

C. Execution on a PR2

The Network Constructor produces an EN representing all the expectations about the execution of the specific plan with respect to the EK, as explained in the previous section.

Currently, we assume that EK specifies the exact coordinates in metric space corresponding to `robotAt(c1)`. Alternatively, the Network Constructor could perform spatial reasoning and the coordinates could be calculated w.r.t. to task, environment and the used platform.

The Execution Monitor needs to establish whether the received observation is relevant to execution, i.e., corresponds to an expected observation. For instance, $tMiddle$ could be the effect of $mTorso$ or and intermediary observation preceding the desired $tDown$. The former may derive from the execution of an `undock` action while the robot is holding an object (recall, in real life the robot cannot lower the torso completely if holding an object as this may collide with its base). The latter may simply be an intermediate state "on the way" to achieving a $tDown$ posture. To ascertain whether the $tMiddle$ observation should be anchored with the corresponding expectation, the EM uses the source (the torso sensor), the value of the observation, and temporal relations expressed in EK. This process is called anchoring, and resembles perceptual anchoring, specifically the Find function [6].

The anchor is added explicitly into the EN as the temporal constraint $observation\{e\}expectation$ (since the two fluents are assumed to represent the same event). The expectations about start or finish time of fleunts connected to some $expectation$ in EN are updated according to the connecting instantiated EK relation. Through these updates, some of the anchored fluents may violate expectations from EK (e.g., a presumed precondition finishes before the corresponding action could be started). In these cases the old anchor is removed and the EM will try establish a new one with a future observation. This behaviour increases the robustness in face of irrelevant or imprecise observations.

The EM also dispatches actions. In absence of explicit temporal relations in EK restricting when to dispatch an action, each action is dispatched immediately when all of its precondition are observed to hold. EK can specify for which observations it makes sense to wait, e.g., an arm posture can be delayed or started/finished prematurely due to sensory noise; conversely, a mug will not appear on the table on its own later and a failure to detect the mug should be treated as a failure. Separating these two cases increases the robustness of execution in face of imprecise and delayed observations.

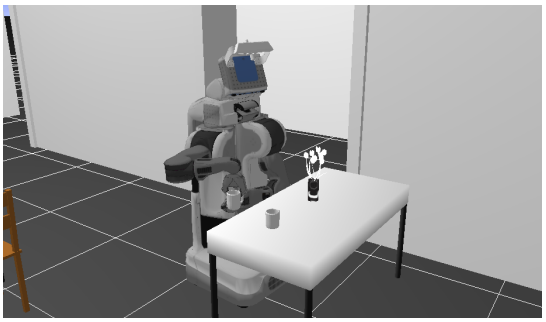

Fig. 4. The PR2 picking up m1

The EN was used to both execute and monitor the mug delivery plan in the GAZEBO simulator with a standard PR2 platform. The `observe` action obtains information about objects from GAZEBO. This means that we get perfect object recognition and identification. The simulated environment contains three tables and four chairs. The Execution Monitor could reliably detect cases of the mug slipping out of the grasp during the execution of `undock`, `drive`, `dock`, `observe`. It also exhibited a degree of robustness against irrelevant and noisy observations of arm postures.

Our approach has been integrated into the architecture of the project RACE [7]. The SHOP2 [9] HTN-planner was used for planning and the MetaCSP-framework [8] was used to implement the EN.

IV. RELATED WORK

PLANEX [2] was arguably the first Plan Execution Monitor. PK is transformed into a *a triangle table* which explicitly expresses the dependency of an action on effects of another action. The authors assumed that PK is directly observable. We do not need such an assumption in our approach, as EK models explicitly the relations between elements of SK and PK. Furthermore, while a triangle table is generated from PK only, we use both PK and EK to produce EN. The PLANEX approach was designed to efficiently find the next action to execute, given the observed SK. Therefore PLANEX could decide to execute repeatedly or skip (unnecessary) parts of the plan. We consider these features part of our future work.

Hierarchical architectures [11], [10], [12] employ translation modules to bridge the gap between high level PK and low level SK. These modules translate instructions from high to low level and translate the resulting low level progress information/error messages back to high which are deliberated upon. As a result, what we call EK is scattered throughout the system, and often not expressed implicitly. This works well with independent actions which can be considered separately, e.g., the possible failure of mug slipping out of grasp would be considered in four separate

actions: `undock`, `drive`, `dock`, `observe`. We avoid this by maintaining an explicit EK.

In [13], Temporal Action Logic (TAL) is used to represent both EK and PK. The DyKnow knowledge processing middleware specifies how derived observations are generated form direct ones. In our approach, this information is explicit in EK and is also used for anchoring/filtering out irrelevant observations. The authors also discuss how EK could be generated automatically from PK and be adapted during execution depending on the currently executed actions. We also generate EK from PK, but we add additional knowledge to EK as well.

In the RoboEarth project [14], a robot can query a database to receive a platform-agnostic "action recipe" (similar to a plan). This recipe is translated into a CRAM plan, a partially ordered set of goals (not actions), reflecting the capabilities of the specific platform. These goals are on the level of abstraction of PK. Procedural information how to achieve the goal or detect progress/failure (based on observations) during execution is delegated to processing modules, and thus remains implicit.

A finite State Machine architecture [15] is often used for Execution Monitoring. The user has to define the states, the transitions and information flow between the states (which loosely corresponds to PK). The states correspond to actions (or sub-plans) and contain (often implicitly) all EK necessary for Execution Monitoring. In our approach, EK is explicit, thus making it easy to adapt and useful for reasoning about dispatching and anchoring.

In [16], a partially ordered plan and a set of temporal constraints is converted into a generalized representation. The executor uses this representation to decide which action to dispatch next. This is similar to how PK is augmented by EK to obtain the EN we use for dispatching. This work was not evaluated on a robot, and so the problem of connecting SK and PK was not addressed.

V. DISCUSSION AND FUTURE WORK

Autonomous execution and execution monitoring of plans on a robot remains challenging. The main problem is how to connect abstract Planning Knowledge (PK) to sensor and actuation information obtained during execution — Situational Knowledge (SK). To this end we propose to use a dedicated Execution Knowledge (EK) formally representing the expectations about future observations in a specific system and their connections to Planning Knowledge (PK).

Since EK is separate from PK, it can use different formalisms and reasoning mechanisms, e.g., qualitative temporal relations as discussed in this paper. EK encapsulates formally specified platform- and environment-specific knowledge. This eases the adaptation of behaviour across platforms (through modifying EK) and the (re)use of platform agnostic PK. EK also contains information absent from PK related to anticipated failures, permissible context of observations and relations between independent actions. For future work we will explore other suitable formalisms to represent EK.

Currently PK, the plan, EK and SK are used to create an Execution Network (EN). The Execution Monitor updates this network during execution based on its current state and SK. A discrepancy between SK and the EN indicates that the environment is in an unexpected state. This state may be also better then expected: some actions may be skipped or altered. The Execution Monitor could detect such opportunities and adapt the EN, the plan or trigger replanning.

If the detected discrepancy is not reflected in PK, replanning is not a feasible strategy. Instead, the EM could leverage EK and perform auxiliary actions observing or altering the system state sufficiently to find a new plan. Another recovery strategy would be to reschedule actions or reconsider an anchor between between an element of PK and SK, where both anchoring and scheduling rely on EK.

ACKNOWLEDGMENT

This work is supported by the EC 7th Framework Program under Project RACE (Robustness by Autonomous Competence Enhancement, grant no. 287752).

REFERENCES

[1] Fikes, R. E.; Nilsson N. J,, STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, in Artif. Intel., vol. 2, no. 3/4, pp. 189-208, 1971
[2] Fikes, R. E., Monitored Execution of Robot Plans Produced by STRIPS, IFIP Congress '71, no. 55, 1971
[3] Helmert, M., Complexity Results for Standard Benchmark Domains in Planning, in Artif. Intel., vol. 143, no. 2, pp. 219-262, 2003
[4] Allen, J.F., Towards a General Theory of Action and Time, in Artif. Intel., vol. 23, no. 2, pp. 123-154, 1954
[5] Š. Konečný; S. Stock; F. Pecora; A. Saffiotti, Planning Domain + Execution Semantics: a Way Towards Robust Execution?, Qualitative Representations for Robots, AAAI Spring Symposium, 2014
[6] S. Coradeschi and A. Saffiotti, An introduction to the anchoring problem, in Robot. Auton. Syst., vol. 43, no. 2, pp. 85-96, 2003.
[7] S. Rockel; B. Neumann; J. Zhang; K. S. R. Dubba; A. G. Cohn; Š. Konečný; M. Mansouri; F. Pecora; A. Saffiotti; M. Günther; S. Stock; J. Hertzberg; A. M. Tomé; A. J. Pinho; L. S. Lopes; S. von Riegen; L. Hotz, An Ontology-based Multi-level Robot Architecture for Learning from Experiences, Designing Intelligent Robots: Reintegrating AI II WS, AAAI Spring Symposium, 2013
[8] F. Pecora, The Meta-CSP Framework: a Java API for Meta-CSP based reasoning, http://metacsp.org, 2013
[9] Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; Yaman, F., SHOP2: An HTN planning system, in J. Artif. Intell. Res., vol. 20, pp. 379-404, 2003
[10] Firby, R. J., Adaptive Execution in Complex Dynamic Worlds, PhD thesis, 1989, Yale University
[11] R. Alami; R. Chatila; S. Fleury; M. Ghallab; F. Ingrand, An Architecture for Autonomy, in Int. J. Robot. R., vol. 17, pp. 315-337, 1998
[12] McGann, C.; Py, F.; Rajan, K.; Ryan, J.; Henthorn, R., Adaptive control for autonomous underwater vehicles, AAAI, 2008
[13] Doherty, P.; Kvarnström, J.; Heintz, F., A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems, Auton. Agent. Multi-Ag., vol. 19, no. 3, pp. 332-377, 2009
[14] di Marco D.; Tenorth M.; Häussermann K.; Zweigle O.; Levi. P, RoboEarth Action Recipe Execution, IAS, 2012
[15] Bohren, J.; Rusu, R.B.; Gil Jones, E.; Marder-Eppstein, E.; Pantofaru, C.; Wise, M.; Mosenlechner, L.; Meeussen, W.; Holzer, S., Towards autonomous robotic butlers: Lessons learned with the PR2, ICRA, pp. 5568-5575, 2011
[16] Muise, Ch.; Beck, J. Ch.; Mcilraith, S. A., Flexible Execution of Partial Order Plans With Temporal Constraints, IJCAI, pp. 2328-2335, 2013
[17] Dechter, R.; Meiri, I.; Pearl, J., Temporal Constraint Networks, in Artif. Intel., vol. 49, no. 1-3, pp. 61-95, 1991
[18] Floyd, R.W., Algorithm 97: Shortest path, in Commun. ACM, vol. 5, no. 16, pp. 345-348, 1962