One Is Not Enough: A Hybrid Approach for IT Change Planning

Sebastian Hagen[1], Nigel Edwards[2], Lawrence Wilcock[2], Johannes Kirschnick[2], and Jerry Rolia[2]

[1] Munich Technical University, Boltzmannstr. 3, 85748 Garching, Germany
hagen@in.tum.de
[2] HP Labs, Long Down Avenue, Bristol BS34 8QZ, United Kingdom
firstname.lastname@hp.com

Abstract. We propose a novel hybrid planning approach for the automated generation of IT change plans. The algorithm addresses an abstraction mismatch between refinement of tasks and reasoning about the lifecycle and state-constraints of domain objects. To the best of our knowledge, it is the first approach to address this abstraction mismatch for IT Change Management and to be based on Artificial Intelligence planning techniques. This has several advantages over previously existing research including increased readability, expressiveness, and maintainability of the descriptions. We developed the foundations of the approach and successfully validated it by applying it to change request planning for *TikiWiki*, a Content Management System.

Keywords: IT Change Management, change planning, policy refinement, AI planning.

1 Introduction

Due to the proliferation of the Software as a Service and the Cloud Computing paradigm, data centers are rapidly growing in size. The complexity of hosted applications increases as well because it has become feasible to host massively distributed applications. This puts additional burden on data center operators and their customers because Change Management becomes more difficult. The generation of change plans is an important step of Change Management as defined by ITIL [7]. Automating the generation of change plans is the key to reduce staff costs, to cope with the complexity of data centers and applications, to comply with company wide policies, and to reduce operator failures. We know of no commercial Data Center Automation product or published research addressing the automated generation of change plans based on workflow descriptions, hierarchical refinement strategies, lifecycle behavior and state-constraints of domain objects. Our work addresses an abstraction mismatch inherent to these descriptions. Planning for change requests (CRs) involves the usage of two contrary abstractions:

C. Bartolini and L.P. Gaspary (Eds.): DSOM 2009, LNCS 5841, pp. 56–70, 2009.

First, reasoning about the *state*, *lifecycle behavior*, and *state constraints* respectively *dependencies* among domain objects is necessary. For example, a database can be in states like *running* or *stopped*. Dependencies refer to the state of other domain objects. For instance, a database can only be installed if the virtual machine is in state *on*. When reasoning about the behavior of domain objects, the notions of state, lifecycle behavior, and state-constraints of domain objects cannot be avoided.

Second, IT change request planning involves the specification of best practice *workflows* and the *refinement of abstract high-level CRs* into finer grained CRs until non-decomposable CRs are reached. This is necessary when planning for abstract high-level CRs. For example, the task to test an application may be decomposed into a set of subtests. This cannot be expressed by lifecycle states of domain objects and constraints among them - an abstraction mismatch.

While this abstraction mismatch has been noted before in the area of Policy Based Management [12], [13], [14], and [11], it has not yet been addressed for IT Change Management. We propose an approach to address this problem for IT change planning. The hybrid approach supports hierarchical task refinement and reasoning about the lifecycle and dependencies of domain objects interchangeably. Its KBs clearly separate hierarchical problem solving strategies, description of domain object behavior, and state-constraints from each other. Thus, domain descriptions produced by an IT practitioner become more readable, extendable, and maintainable because the abstractions are clearly separated and made explicit. The separation also paves the way for a simple methodology to write KBs. The remainder of this paper is organized as follows: Section 2 introduces the basic terms underlying the hybrid approach. The contributions are highlighted in Sect. 3. Section 4 introduces the algorithm based on a TikiWiki planning example. The performance of our prototype is evaluated in Sect. 5. Related Work concerning IT change planning and Policy Based Management is discussed in Sect. 6. Finally, Sect. 7 concludes the paper.

2 Conceptual Model and TikiWiki Planning Domain

This section describes the three levels comprising the conceptual model of our approach. The Domain Entity level in Subsect. 2.1 provides an object oriented model to reason over. The Behavioral level describes the behavior of domain objects. It is introduced in Subsect. 2.2. Finally, we explain the Refinement level which describes workflows and task decomposition rules.

2.1 Domain Entity Level

The Domain Entity level defines two models, the *Domain Object Model (DOM)* and the *Domain CR Model*. The *Domain Object Model* provides an object oriented model representing the infrastructure and hosted software components. An instance of the DOM is used during planning to evaluate preconditions and to make effects of planned operations persistent. The DOM can be based on

modeling techniques like CIM or EMF. We are using Groovy [6] because it is Java compatible, supports configuration templates to instantiate OO models, and Domain Specific Languages (DSL) can be easily written. In the DOM used throughout this work, a TikiWiki cloud service consists of a database, several Apache servers, and one load balancer. It is shown in Fig. 2.

Groovy methods implemented in the DOM can be called by the planner to implement the effects of actions. The DSL describing the planning domain becomes more readable when complex change behavior can be hidden behind method calls. This feature is not supported by traditional Artificial Intelligence (AI) planners based on predicate KBs, e. g., SHOP2 [9]. This also makes it easier to use our approach with legacy models. We assume that the state of a domain object is automatically kept updated by the DOM's methods. Note that the DOM can be easily extended to take physical infrastructure into account.

Fig. 1. eSTS for domain objects

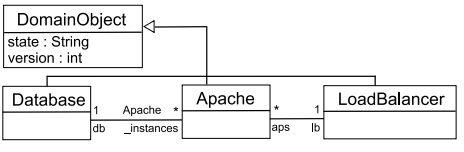

Fig. 2. DOM of TikiWiki domain

The *Domain CR Model* is a model for the different kind of CRs. A change request cr is a 6-tuple $cr = (n_{cr}, o_{cr}, params_{cr}, HB_{cr}, parent_{cr}, t_{cr})$, where n_{cr} is the name of cr, o_{cr} the target of the CR, i. e., an instance of class *DomainObject* in the DOM, affected by cr, $params_{cr}$ a map of parameters customizing cr, and HB_{cr} a set of CRs happening before cr. Throughout this work HB_{cr} is considered to be non-transitive. The HB sets of all CRs can be used to compute the transitive closure of all CRs happening before a particular CR. $parent_{cr}$ is the parent of cr. $t_{cr} \in \{na, at, sc\}$ denotes the type of a CR with the following characteristics: A *non-atomic (na)* CR is subject to further refinement into child CRs by a method defined in the Refinement level. An *atomic (at)* change request cannot be further refined. It has effects on the DOM described by operators in the Refinement level. A *state-changing (sc)* CR is subject to further refinement; All of it descendants contribute to the state change in domain object o_{cr}, e. g., to resolve dependencies or to actually perform its state-change.

2.2 Behavioral Level

The *Behavioral level* describes the lifecycle-behavior of domain objects by means of *extended restricted state-transition systems* (eSTSs), an extended version of restricted STSs [4]. To keep matters simple in our modeled domain, we assume that databases, Apache instances, and load balancers have the same eSTS. A simplified graphical representation of it is given in Fig. 1. More formally, eSTS σ is a 3-tuple $\sigma = (p_\sigma, S_\sigma, T_\sigma)$ such that p_σ is a precondition evaluated over

a domain object o to decide whether σ describes the behavior of o. See the boolean expression in Line 1, List. 1 which associates the eSTS to every instance of class *DomainObject*. S_σ is the set of states (see Line 2 in List. 1) and T_σ the set of transitions of σ (see Lines 3–13). A transition $t \in T_\sigma$ is a 5-tuple $t = (s_t, g_t, cr_t, D_{t\_pre}, D_{t\_post})$ such that $s_t \in S_\sigma$ is the source and $g_t \in S_\sigma$ the sink of transition t. Line 4 in Listing 1 defines the *stop* transition from the *Started* to the *Installed* state. cr_t is the CR that needs to be achieved to change the state from s_t to g_t. For example, Line 5 in List. 1 links a CR called *stop*, i. e., $n_{cr} == stop$, to transition *stop*. D_{t_pre} and D_{t_post} are dependencies that need to be fulfilled before or after cr_t is planned for, i. e., before or after the transition is taken in the eSTS. $D_{t\_pre} = (p_{t\_pre}, CRS_{t\_pre})$ where p_{t_pre} is a precondition evaluated over a domain object to determine whether the dependency CRs described in CRS_{t_pre} need to be planned for before cr_t. For instance, the precondition of the dependency given in Lines 7–12 in List. 1 says that it is only valid for an instance of Database. The dependency CRs (CRS_{t_pre}) of the dependency are defined in Lines 11–12. They describe state-changing CRs to stop all Apache servers, i. e., to change their state to *Installed*. Thus, the dependency CRs are only planned for if the stop transition in a database is taken. If the stop transition in an Apache server is taken, then the depedency does not apply because the precondition evaluates to false. All in all, the DSL given in Listing 1 describes the eSTS linked to all domain objects, its three states, the *stop* transition, the *stop* CR linked to the transition, and a dependency to stop all Apache instances before the stop transition is executed in a database. The other transitions and dependencies were omitted due to space constraints.

Listing 1. DSL for eSTS

```
1  eSTS(it.instanceof DomainObject) {
2   states {["Removed","Installed","Started"]}
3   transitions {
4    transition ("stop", from : "Started", to : "Installed"){
5     subtask {stop target : it.target}
6     dependencies {
7      dependency{
8       precondition{it.target instanceof Database}
9       type{time : "before", decomp : "parallel"}
10      subtasks{
11       it.target.Apache_instances.each{
12        set-state target : it, goal_state : "Installed"
13       }}}...}}...}
14 }
```

Note that the dependency in List. 1 only applies for a database (Line 8) while a database has the same states and transitions as every other domain object. The DSL decouples the dependency specification from the lifecycle specification to specify domain objects with the same lifecycle but with different dependency behavior within one *eSTS*. This increases the usability and reuseability of the Knowledge Base. The code in Line 11 in List. 1 directly navigates the DOM instance to create the dependency CRs (CRS_{t_pre}) of transition *stop*. *it* refers to the state-changing CR which the planner plans for using the eSTS the dependency

is specified within. By following the *target* reference the database domain object (o_{cr}) of the DOM instance is reached holding a list of all Apache instances (see *Apache_instances* reference in Fig. 2). We only need to specify dependencies that directly affect other domain objects when executing a transition. For example, the eSTS holds a before-dependency to stop the Apache servers before the database is stopped (Lines 7–13). The eSTS describing Apache, in this case the same eSTS, holds a before-dependency for Apache instances to stop the load balancer before transition *stop* is taken. This is how we can exploit transitivity over dependencies when stopping a database. Note that the latter dependency is only valid if the Apache to stop is the last one running in the system. Such constraints can be defined in the precondition (p_{t_pre} or p_{t_post}) of a dependency.

2.3 Refinement Level

The *Refinement level* describes rules for CR decomposition, workflows, and best practice problem solving strategies not expressible by eSTSs and dependencies. A *(Hierarchical Task Network (HTN)) method* m [3], [4] describes a sequential or parallel decomposition of a non-atomic CR into child CRs. More formally let $m = (n_m, p_m, CRS_m)$ be a 3-tuple where n_m is the name of m, p_m is a precondition to determine whether m is applicable, CRS_m is a set of child CRs to be achieved in order to achieve the CR m is applied to. The children can either be in no temporal relationship, i. e., $\forall cr \in CRS_m : HB_{cr} = \emptyset$, or they can be in a sequential relationship such that $HB_{cr_{n+1}} = \{cr_n\}$. Let cr be a non-atomic CR, i. e., $cr = (n_{cr}, o_{cr}, params_{cr}, HB_{cr}, parent_{cr}, na)$, then m can be applied to decompose cr iff $n_{cr} == n_m$ and p_m is satisfied by o_{cr}, i. e., their names match and the precondition evaluates to true over the target of cr.

More practically, patching a domain object is defined by the following workflow: *(CR1)* The domain object to patch is stopped. *(CR2)* The update is applied. *(CR3)* The domain object is tested. *(CR4)* The previous state of the patched domain object is restored. The decomposition of *patch* incorporates state-changing CRs (1 and 4) and non-atomic/atomic CRs (2 and 3). It is tasks like patching and testing where the hybrid approach can play out its advantages because planning works interchangeably between refinement and state-transition systems. The KB engineer can rely on the previously defined behavior and dependencies in the Behavioral level when writing methods.

Listing 2. DSL for *patch* method

```
1 method(name : "patch", pre : it.target instanceof DomainObject) {
2  subtasks {
3   sequential {
4    String state_old = it.target.state
5    set-state target : it.target, goal_state : "Installed"
6    update target : it.target
7    test target : it.target
8    set-state target : it.target, goal_state : state_old}}}
```

Listing 2 shows the DSL for the *patch* method. Line 1 describes the name n_m (patch) and the precondition p_m (target of CR, i. e., o_{cr}, is instance of class *DomainObject*). Lines 2–8 describe CRS_m as the sequential decomposition of the *patch CR* into CRs 1-4 as Groovy code. Line 4 saves the current state of o_{cr} in the local variable *state_old*. Lines 5 and 8 describe state-changing CRs to stop (CR1) and to restore (CR4) the previous state of the target of the patch CR (o_{cr}). The previously locally defined variable *state_old* is used as the *goal_state* parameter of the last state-changing CR to restore the domain object's old state in Line 8. Intermediate CRs with names *update* and *test* need to be planned for. *update* is an atomic CR, i. e., a non-decomposable CR with effects on the DOM. Atomic CRs are implemented by *(HTN) operators* [3], [4] also specified in the Refinement level. More formally, an operator o is a 3-tuple $o = (n_o, p_o, e_o)$. n_o is the name of o. See Line 1, Listing 3 for an operator named *update*. p_o is a precondition to determine whether o can be applied and e_o describes the effects of o. Let $cr = (n_{cr}, o_{cr}, params_{cr}, HB_{cr}, parent_{cr}, at)$ be an atomic CR. o is applicable to cr if $n_{cr} == n_o$ and p_o is satisfied by o_{cr}. In case of *update* the precondition p_o is specified in Lines 1–2 in Listing 3. It demands that o_{cr} is an instance of class *DomainObject* and currently in state *Installed*. The effects e_o are defined by the programmatic Groovy code in Lines 4–5. The code is executed by the planner, increasing the version attribute of o_{cr} (Line 4) and returning true to signal the successful execution to the planner. Note that exceptions from the DOM instance can be caught to return false to the planner to trigger backtracking. A method to refine *test* into subtests is not given because with *patch* an exemplary method has already been provided.

Listing 3. DSL for *update* operator

```
1 operator(name : "update", pre : it.target instanceof DomainObject
2  && it.target.state == "Installed") {
3    effects {
4     it.target.version++
5     return true
6 }}
```

Having defined the Behavioral level previously, there is no need to worry about dependencies and the behavior of domain objects. Behavior described in eSTSs (see List. 1) is strictly separated from refinement strategies described by methods (see List. 2) and operators in List. 3. Both abstractions can be written by different KB engineers. One an expert in the lifecycle-management of applications and the the other one an expert in best practices in Change Management workflows. Linking Behavioral and Refinement level together overcomes the abstraction mismatch enabling IT practitioners to more naturally express the planning domain. The clear separation between Refinement and Behavioral level makes it easier to write and change domain descriptions. For example, to extend the behavior of a domain object states and transitions can be added. The refinement strategies stay unchanged. Vice versa, the same applies when changing workflows.

3 Contributions of the Hybrid Approach

Task refinement and state-based reasoning with constraints are important for IT change request planning. The hybrid approach brings both abstractions together. Previous work regarding IT change request planning has been focused on either of the two abstractions. Only supporting Refinement as done by HTN Planners [3], [9] is powerful enough to plan for CRs, but has several drawbacks. Without a Behavioral level, eSTSs have to be described by methods in the Refinement level. For instance, the eSTS shown in Fig. 1 can be in 3 states and any state-changing CR regarding this eSTS can have two goal states different from the current state. Thus, $3 * 2 = 6$ methods need to be written as a replacement for state-changing CRs. In general, $n * (n - 1)$ methods have to be written for a domain object with n states. These 'state-changing' methods will be mixed with higher-level workflow methods like the *patch* method. Furthermore, additional HTN methods are needed to describe different dependency behavior for domain objects with the same lifecycle. All in all, the KB becomes more difficult to read and to maintain because concepts are mixed. Other refinement approaches [1], [2] restrict refinement to be based on dependencies. Workflows not driven by dependencies, e. g., restarting an application, are difficult to describe in such an approach.

Only providing a Behavioral level [5] and no refinement capabilities reduces expressiveness. For instance, the *patch* and *test* CRs cannot be planned for only based on state-constraints. There is no convenient way to express that subtests need to be done by using eSTSs and state-constraints. It is more natural to express this as a refinement rule. Even if refinement could be described using eSTSs, dependencies, or state-constraints, the drawback that refinement is mixed with eSTSs or constraints still exists. The hybrid approach offers native support for both abstractions, clearly distinguishes between them, and keeps them separated in the Knowledge Base. However, reasoning about refinement and states can naturally refer to each other. This offers *two advantages* when comparing it to previous work in IT change planning. *First*, to the best of our knowledge it is the first approach to *natively support workflow-, task-refinement-, and state-based-planning based on constraints* at the same time. *Second*, *KBs of the hybrid approach are easier to maintain and to read* because the abstractions are described in their most natural way and are separated from each other. To the best of our knowledge, it is also the first work to apply findings from AI planning to IT change planning. HTN planning [3], [4] and our extensions to plan for state-changing CRs reason about the effect a CR has on the DOM. Plans are sound from a computational point of view if the algorithm, methods, operators, and eSTSs are sound. We leave this proof to future work. We also show that planning can be done with reasonable overhead over an OO model. This bridges the gap between AI planning and IT Change Management because models like CIM have been traditionally used to represent the IT system to manage.

4 The Algorithm

The algorithm takes a 4-tuple $(M, O, \sum, Q)$ as input where M is a set of methods, O a set of operators, $\sum$ a set of eSTSs, and Q a queue of CRs to plan for. To plan for the *patch* CR, Q holds the ordered set $\{cr_1\}$ where $cr_1 = (patch, db, [:], \emptyset, null, na)$ at the initial call to the algorithm. See Fig. 3 for the decomposition tree with root node cr_1 and all of its descendants created during planning. Note that $o_{cr_1} == db$, i. e., the database is to be patched. The underlying DOM instance consists of one database (db), one Apache server (ap), and one load balancer (lb). All of them are in state *Started*.

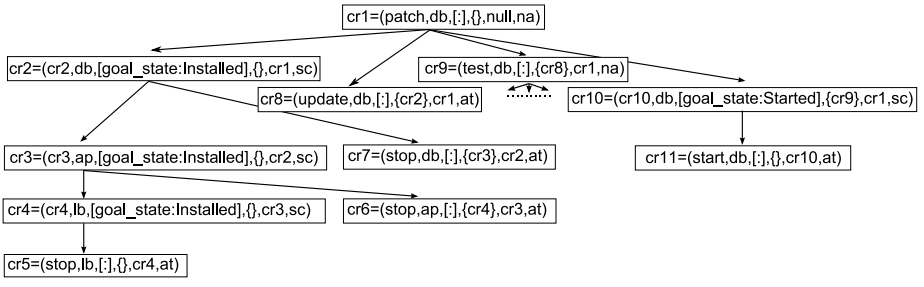

Fig. 3. Decomposition tree for *patch* CR

4.1 Decomposing Non-atomic CRs

Algorithm 1 describes the decomposition of non-atomic CRs, e. g., cr_1. A set of applicable methods to decompose cr_1 is determined (Line 2) as described in Subsect. 2.3. Only one method m, the method previously provided in List. 2, is applicable. It is chosen in Line 3. CRS_m, the list of child CRs described by m, is $\{cr_2, cr_8, cr_9, cr_{10}\}$. Note that each cr of these has the preceding sibling in its HB_{cr} set to formalize the sequential relationship among the children. See the *patch* method in List. 2 and the tree in Fig. 3 for temporal constraints. For each descendant the parent is set to cr_1 in Line 5. Finally, cr_1 is removed from the queue Q and CRS_m are added at the front, leading to $Q = \{cr_2, cr_8, cr_9, cr_{10}\}$.

4.2 Decomposing State-Changing CRs

cr_2 is extracted from Q and passed to Algorithm 3. Line 2 determines a matching eSTS for db, the target (o_{cr_2}) of cr_2. See Subsect. 2.2 for the matching process. We assume that there always exists exactly one eSTS σ mapped to a domain object o. The goal state (*Installed*) to be achieved in σ is extracted from the parameters of cr_2 and c is initialized with the current state of db (*Started*) in Line 3. T' holds the path from c to g (Line 4). We assume that the path between every state-pair is unambiguous. The path only consists of transition *stop*, thus $T' = \{t\}$ where $t = (Started, Installed, stop, D_{t\_pre}, D_{t\_post})$. Some helper variables

if $cr == (n_{cr}, o_{cr}, params_{cr}, HB_{cr}, parent_{cr}, na)$ **then**
  Be $M' = \{(n_m, p_m, CRS_m) \in M | p_m$ is satisfied over $o_{cr} \ \wedge \ n_m == n_{cr}\}$
  Choose non-deterministically $m \in M'$
  foreach $cr' \in CRS_m$ **do**
    $parent_{cr'} = cr$
  end
  $Q = CRS_m \circ rest(Q)$
end

Algorithm 1. Planning for non-atomic CRs

if $cr == (n_{cr}, o_{cr}, params_{cr}, HB_{cr}, parent_{cr}, at)$ **then**
  Be $O' = \{(n_o, p_o, e_o) \in O | p_o$ is satisfied over $o_{cr} \ \wedge \ n_o == n_{cr}\}$
  Choose non-deterministically $o \in O'$
  if *effects* e_o *are succesfully applied to* o_{cr} **then** $Q = rest(Q)$ **else** backtrack()
end

Algorithm 2. Planning for atomic CRs

if $cr == (n_{cr}, o_{cr}, params_{cr}, HB_{cr}, parent_{cr}, sc)$ **then**
  Be $\sigma = (p_\sigma, S_\sigma, T_\sigma) \in \sum$ a eSTS such that p_σ is satisfied over o_{cr}
  Let $g = params_{cr}$[goal_state]; Let c be the current state of o_{cr}
  Let $T' \subseteq T_\sigma$ be an ordered set of transitions leading from c to g in σ
  $cr_{t\_old} = null$; $CRS_{post\_old} = null$; $children = []$
  foreach $t == (s_t, g_t, cr_t, (p_{t\_pre}, CRS_{t\_pre}), (p_{t\_post}, CRS_{t\_post})) \in T'$ **do**
    if p_{t_pre} *evaluates to* $false$ *over* o_{cr} **then** $CRS_{t\_pre} = \emptyset$
    if p_{t_post} *evaluates to* $false$ *over* o_{cr} **then** $CRS_{t\_post} = \emptyset$
    $children = children \circ CRS_{t\_pre} \circ [cr_t] \circ CRS_{t\_post}$
    $HB_{cr_t} = HB_{cr_t} \cup CRS_{t\_pre}$
    $\forall cr' \in CRS_{t\_post} : HB_{cr'} = HB_{cr'} \cup [cr_t]$
    if $cr_{t\_old} \neq null \wedge CRS_{post\_old} \neq null$ **then**
      switch $(CRS_{post\_old}, CRS_{t\_pre})$ **do**
        case $(\emptyset, \emptyset) : HB_{cr_t} = HB_{cr_t} \cup [cr_{t\_old}]$
        case $(\emptyset, \neq \emptyset) : \forall cr' \in CRS_{t\_pre} : HB_{cr'} = HB_{cr'} \cup [cr_{t\_old}]$
        case $(\neq \emptyset, \emptyset) : HB_{cr_t} = HB_{cr_t} \cup CRS_{post\_old}$
        case $(\neq \emptyset, \neq \emptyset)$:
          $\forall cr' \in CRS_{t\_pre} : HB_{cr'} = HB_{cr'} \cup CRS_{post\_old}$
      end
    end
    $cr_{t\_old} = cr_t$; $CRS_{post\_old} = CRS_{t\_post}$
  end
  $\forall cr' \in children : parent_{cr'} = cr$; $Q = children \circ rest(Q)$
end

Algorithm 3. Planning for state-changing CRs

are initialized in Line 5. After that, Lines 6–22 iterate over the transitions in path T'. p_{t_pre} evaluates to true over db in Line 7. p_{t_pre} is defined in Line 8 in List. 1. The precondition for the dependency applies because t is taken within a database. Thus, $CRS_{t\_pre} == \{cr_3\}$ stays untouched. If p_{t_pre} did not apply, the dependency tasks in CRS_{t_pre} are erased because they do not become children (Line 9). The same is done regarding post-dependencies of t. There is no post-dependency, thus p_{t_post} is false. Line 9 builds the children comprising transition t. In our case $children = []\circ[cr_3]\circ[cr_7]\circ[] = [cr_3, cr_7]$. The HB sets of the children need to be adapted. All pre-dependency CRs, i. e., the CRs in CRS_{t_pre}, need to happen before cr_t (cr_7) (Line 10). In our case $HB_{cr_7} = \emptyset \cup \{cr_3\} = \{cr_3\}$. Line 11 adapts the HB relation in every post-dependency CR because cr_t happens before them. Nothing is done because $CRS_{t\_post} == \emptyset$. The *if* statement in Lines 12–20 checks whether t is at least the second transition taken in T'. As T' only consists of t the body is not executed. For two consecutive transitions $t_1 \in T'$ and $t_2 \in T'$, the body of the *if* statement is executed to adapt the HB sets of CRs in cr_{t_1}, $CRS_{t_1\_post}$, $CRS_{t_2\_pre}$, and cr_{t_2} to reflect that CRs of t_2 need to happen after CRs of t_1.

Four cases, depending on the values of CRS_{post_old}, the post-dependencies of the previous transition, and CRS_{t_pre}, the pre-dependencies of the current transition, are distinguished using pattern matching in the *switch* statement in Lines 13–19. *(1)* Both sets are empty. This means cr_{t_1} and cr_{t_2} directly follow each other. Thus, cr_{t_old}, storing the task linked to the previous transition, is added to HB_{cr_t} (Line 14). *(2)* The previous transition did not have post-dependencies and there are pre-dependencies for the current transition. In this case, the task linked to the previous transition (cr_{t_old}) needs to happen before each cr $\in CRS_{t\_pre}$. *(3)* $CRS_{post\_old} \neq \emptyset$ and $CRS_{t\_pre} == \emptyset$, i. e., cr_t directly follows the CRs in CRS_{post_old}. Every CR in CRS_{post_old} needs to happen before cr_t. Thus, CRS_{post_old} is added to HB_{cr_t} in Line 16. *(4)* Both sets are non-empty. Every post-dependency CR of the previous transition needs to happen before every pre-dependency CR of the current transition (Line 18). Line 20 saves cr_t in cr_{t_old} and the post-dependencies of t in CRS_{post_old} to refer to them in the next iteration when setting dependencies. Finally, the parent of the children is set to cr and they are added at the front of the queue in Line 23. Q now holds $\{cr_3, cr_7, cr_8, cr_9, cr_{10}\}$. Planning continues with cr_3 and cr_4 which are similarly decomposed as cr_2.

4.3 Planning for Atomic CRs

cr_5 is the first atomic CR to be planned for. Algorithm 2 determines all applicable operators for cr_5 in Line 2. Applicability of operators is defined in Subsect. 2.3. We assume that there exists an applicable operator for the *stop* CR. It's effects are applied to o_{cr_5}, lb, in Line 4. For example, the operator could call a *stop* method on lb. If the operator returns true, cr_5 is removed from Q and planning continues with cr_6 and cr_7 (see Fig. 3) applying the same operator to them. If the execution of the operator failed, i. e., it returns false, backtracking is triggered. Backtracking finds a previously planned CR, that has another decomposition alternative and restarts

planning from that CR onwards to generate a non-failing decomposition. The next CR in Q to plan for is cr_8, the atomic *update* CR. The operator previously given in List. 3 can be applied because its precondition is satisfied by *db*. The operator increases the *version* attribute of *db*. Planning continues with cr_9 the non-atomic CR to execute the tests. We do not further elaborate on its decomposition because we already showed how non-atomic CRs are decomposed based on cr_1. cr_{10}, a state-changing CR to start *db*, is next in Q. As cr_2 left *db* in state *Installed*, only transition *start* (t) needs to be taken. There are no depedencies associated to t, thus cr_{10} is only decomposed into cr_t, an atomic *start* task (cr_{11}). Planning ends with the application of an operator to cr_{11}.

4.4 Handling Conflicts among CRs

CRs are always planned for in sequential order. Two CRs happening in parallel could target the same domain object leading to unsound plans if their effects interfere and an execution engine does not execute them the order they were planned for. An additional temporal constraint needs to be added to keep the plan sound. Let HB'_{cr} be the set of all CRs planned and happening before cr. $parent_{cr}$ denotes the parent of cr, $descendants_{cr}$ the set of all descendants of cr, and HB_{cr} the local happening before relation of cr. HB'_{cr}, the multiset of all CRs planned and happening before cr, is defined as follows:

$$HB'_{cr} = HB_{cr} \cup \bigcup_{cr' \in HB_{cr}} descendants(cr') \cup HB'_{parent_{cr}} \cup \bigcup_{cr' \in HB_{cr}} HB'_{cr'}.$$

Less formally, HB'_{cr} consists of all CRs directly happening before cr (HB_{cr}), all descendants of these CRs ($\bigcup_{cr' \in HB_{cr}} descendants(cr')$), all CRs beeing planned for and happening before the parent $HB'_{parent_{cr}}$, and of all CRs transitively happening before cr ($\bigcup_{cr' \in HB_{cr}} HB'_{cr'}$). Let Ω_{cr} be the set of CRs already part of the decomposition tree before planning for cr. Let $anchestors_{cr}$ be the set of all anchestors of cr. Then HP_{cr}, the set of all CRs previously planned for and happening in parallel to cr, is defined as $HP_{cr} = \Omega_{cr} - \{HB'_{cr} \cup anchestors_{cr}\}$. Using HP_{cr} rules can be defined to resolve conflicts between parallel CRs. Two constraints need to be checked before planning for a CR cr:

- If there exists a $cr' \in HP_{cr}$ with the same target, i. e., $o_{cr} == o_{cr'}$, then cr needs to happen after cr', i. e., $HB_{cr} = HB_{cr} \cup [cr']$. This prevents parallel execution of cr and cr' by an execution engine. Our approach assume that CRs targeting different domain objects can be executed in parallel.
- If cr is a state-changing CR and a state-changing CR $cr' \in anchestors_{cr}$ exists, such that $o_{cr'} == o_{cr}$, then planning needs to fail because an anchestor CR of cr already tries to change the state of the same domain object.

5 The Prototype

This section introduces and evaluates the performance of the developed prototype. Our experiments were conducted using WinXP, an Intel Xeon with 3Ghz, and 1GB of RAM. The planner ran in non-Gui mode to evaluate the performance

of the pure algorithm. We planned for the *patch* CR using our fully developed KB of the TikiWiki domain. A small underlying DOM instance comprising one database, three Apache servers, and one load balancer produces a plan (the atomic leaf nodes of the decomposition tree) consisting of 64 atomic CRs. In total, the planner had to plan for 153 CRs (30 na, 64 at, and 59 sc) to completely decompose the *patch* CR. The depth of the decomposition tree is 11. Planning took 3.6 seconds and 1,900 domain objects were serialized. Serialization of domain objects is necessary to restore older instances of the DOM in case the planner backtracks. With increased size of the DOM planning takes longer. Planning for 10 Apache servers results in a decomposition tree with 419 CRs, a plan comprising 176 atomic CRs, and 14,000 domain object serializations. Total planning time is 14.3 seconds, whereas 6.7% are spent on serialization consuming 1.6 MB of main memory.

The serialization and deserialization mechanism does not turn out to be a significant bottleneck. For instance, serializing 100,000 large domain objects takes 5.5 seconds and consumes 38.5 MB of main memory. Deserialization on backtracking is slightly slower because the target reference has to be rerouted to the restored domain object for each CR of the decomposition tree. The prototype serializes the DOM instance before planning for any CR. Instead, it is sufficient to serialize the DOM instance only for atomic CRs because only these can change the model. To restore the model for a non-atomic or state-changing CR the DOM instance associated to the latest planned atomic CR needs to be restored. In case of the *patch* example with three Apache instances 48% less domain objects are serialized. All in all, the prototype proves that the generation of large plans is possible within reasonable time compared to the actual execution time of the plan.

6 Related Work

This section discusses related work in IT change planning, policy refinement, and AI planning. *CHAMPS* by Keller et al. [8] formalizes planning and scheduling as an optimization problem achieving a high degree of parallelism. Planning is solely based on dependencies. CHAMPS does not address the abstraction mismatch. Compared to our approach, the algorithm does not reason about the effects of actions. It is difficult to reason about interaction of actions not previously ascertained in a dependency structure. CHAMPS includes scheduling and domain descriptions are automatically derived, while we focus on planning and rely on an IT practitioner to describe the domain. Different to CHAMPS, we focus on state-related constraints. However, we successfully planned for non-state related constraints using methods and suitable preconditions.

Cordeiro et al. [1] introduce the notion of templates to reuse knowledge in IT change design. Plan templates can be described by methods in our approach. Different to CHAMPS, their algorithm plans for task refinement based on dependencies. Compared to our work, the lifecycle-behavior and state-constraints of domain objects are not made explicit. Furthermore, refinement not based on

dependencies is difficult to describe. Similar to CHAMPS, effects of actions are not taken into account.

Aware of this, an algorithm that takes the effects of actions into account is later proposed in [2]. Compared to our solution refinement of tasks is solely based on dependencies, no native support for state-constraints is given, and lifecycle-behavior is not made explicit. In addition to that, interchangeable reasoning about refinement not necessarily based on dependencies and lifecycle-behavior is not readily possible. Thus, Cordeiro et al. do not address the abstraction mismatch. The work focuses on refinement driven by constraints.

Goldsack et al. [5] argue in favor of a pure declarative, state-based approach to manage large data centers. Workflows are considered harmful due to side effects, concurrent changes, and their procedural instead of declarative nature. Similar to CHAMPS reasoning by refinement is not supported. Different to Goldsack et al. our solution supports refinement.

Also related to our work is the area of *Policy Based Management* [10], in particular policy refinement. It is generally concerned with the refinement of abstract high-level *Quality of Service* (QoS) goals into lower-level policies to achieve the higher-level policy. There are subtle differences between IT change planning and policy refinement. Traditionally, plans generated in IT Change Management do not describe alternatives and are very much driven by best practices. Different to that, change plans in policy refinement are goal driven and tend to consist of *on event do something* policies that can be even stored for later evaluation. Thus, plans to achieve a high-level policy tend to incorporate different alternatives or might be adapted when events occur.

Recent work on policy refinement has taken the abstraction mismatch into account. Bandara et al. [12], [13] propose a Goal-oriented policy refinement technique based on the Event Calculus and abductive reasoning techniques. High-level goals formalized in temporal logic are refined until System Goals are reached using formal refinement patterns introduced by Darimont et al. [11]. System Goals can be implemented by state changes (called a strategy). Bandara et al. [12], [13] use abductive reasoning, while Rubino-royal et al. [14] propose to use model checking to derive strategies. Compared to our work, reasoning is proven to be sound but is limited to a small set of goal decomposition patterns [11]. We do not use abductive reasoning [13] or model checking [14] to derive a strategy but state exploration of the STS. In [12], [13], and [14] state changes to domain objects do not trigger further refinement steps. Low-level goals always address a state to be achieved. This is different to our approach, where operators do not need to change the lifecycle-state of a domain object. Similar to their work, we use transition systems to describe the behavior of domain objects.

Also related to our work is the area of AI planning, particularly HTN planning, which is used in the Refinement level. It has been subject to intense research [4]. It does not natively support to reason about the lifecycle-behavior of domain objects, justifying the research regarding the hybrid approach. To the best of our knowledge HTN has not yet been applied to IT change planning.

7 Conclusion and Future Work

We have identified an abstraction mismatch between task refinement, reasoning about the lifecycle of domain objects, and state-constraints in the area of IT change request planning. We have proposed a hybrid algorithm to natively address these abstractions and to reason interchangeably about them. Our results are quite positive. Having previously written KBs for IT change planning in SHOP2 [9], a pure HTN planner, we found it easier to write and to maintain KBs for the hybrid approach due to the clear separation of concepts. Our prototype proves that the algorithm can generate larger plans within reasonable time and that OO models can be used as KBs to be planned over with reasonable overhead for model backup and restoration.

For future work we envision a hybrid algorithm using a declarative specification of state constraints and task refinement to address the abstraction mismatch. Additional value could be generated because thinking declaratively about state-constraints is easier than the procedural like approach we explored in this paper. Furthermore, it provides us with more freedom to specify state-changing CRs affecting many domain objects. We envision a Groovy based language inspired by first-order-logic to qualify and quantify domain objects of the Domain Entity level that are affected by state constraints. In addition to that, we want to examine scalability, replanning, optimization, and scheduling techniques regarding the hybrid approach.

References

1. da Costa Cordeiro, W.L., Machado, G.S., Daitx, F.F., et al.: A template-based solution to support knowledge reuse in IT change design. In: Proceedings of Network Operations and Management Symposium 2008, pp. 355–362 (2008)
2. da Costa Cordeiro, W.L., Machado, G.S., Andreis, F.G., Santos, A.D., Both, C.B., Gaspary, L.P., Granville, L.Z., Bartolini, C., Trastour, D.: A Runtime Constraint-Aware Solution for Automated Refinement of IT Change Plans. In: De Turck, F., Kellerer, W., Kormentzas, G. (eds.) DSOM 2008. LNCS, vol. 5273, pp. 69–82. Springer, Heidelberg (2008)
3. Erol, K., Hendler, J., Nau, D.S.: HTN planning: Complexity and expressivity. In: Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI 1994), vol. 2, pp. 1123–1128. AAAI Press/MIT Press, Washington (1994)
4. Ghallab, M., Nau, D.S., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann, San Francisco (2004)
5. Goldsack, P., Murray, P., Farrell, A., Toft, P.: SmartFrog and Data Centre Automation. In: HPL Technical Report, HPL-2008-35 (2008), `http://www.hpl.hp.com/techreports/2008/HPL-2008-35.pdf`
6. Groovy Homepage, `http://groovy.codehaus.org/`
7. IT Infrastructure Library: ITIL Service Transition (V3), `http://www.itil-officialsite.com/home/home.asp`
8. Keller, A., Hellerstein, J.L., Wolf, J.L., Wu, K.-L., Krishnan, V.: The CHAMPS system: change management with planning and scheduling. In: Proceedings of Network Operations and Management Symposium (NOMS 2004). IEEE/IFIP, pp. 395–408 (2004)

9. Nau, D., Muoz-avila, H., Cao, Y., Lotem, A., Mitchell, S.: Total-order planning with partially ordered subtasks. In: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001), pp. 425–430 (2001)
10. Boutaba, R., Aib, I.: Policy-based Management: A Historical Perspective. Journal of Network and Systems Management, 5(4), 447–480 (2007)
11. Darimont, R., Lamsweerde, A.: Formal refinement patterns for goal-driven requirements elaboration. ACM SIGSOFT Software Engineering Notes 21(6), 179–190 (1996)
12. Bandara, A.K., Lupu, E.C., Moffet, J., Russo, A.: A Goal-based Approach to Policy Refinement. In: Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004), Washington, DC, USA, p. 229. IEEE Computer Society, Los Alamitos (2004)
13. Bandara, A.K., Lupu, E.C., Russo, A., Dulay, N., Sloman, M., Flegkas, P., Charalambides, M., Pavlou, G.: Policy refinement for DiffServ quality of service management. In: Proceedings of 9th IFIP/IEEE International Symposium on Integrated Network Management (IM 2005), pp. 469–482 (2005)
14. Rubio-Loyola, J., Serrat, J., Charalambides, M., Flegkas, P., Pavlou, G., Lafuente, A.L.: Using linear temporal model checking for goal-oriented policy refinement frameworks. In: Proceedings of Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005), pp. 181–190 (2005)