Self-Adaptation with End-User Preferences:
Using Run-Time Models and Constraint Solving*

Hui Song[1], Stephen Barrett[1], Aidan Clarke[2], Siobhán Clarke[1]

[1]Lero: The Irish Software Engineering Research Centre
SCSS, Trinity College Dublin, College Green, Dublin 2, Ireland
[2]Software Group, IBM Ireland, Dublin 15, Ireland
firstname.lastname@scss.tcd.ie, aidan_clarke@ie.ibm.com

Abstract. This paper presents an approach to developing self-adaptive systems that takes the end users' preferences into account for adaptation planning, while tolerating incomplete and conflicting adaptation goals. The approach transforms adaptation goals, together with the run-time model that describes current system contexts and configurations, into a constraint satisfaction problem. From that, it diagnoses the conflicting adaptation goals to ignore, and determines the required re-configuration that satisfies all remaining goals. If users do not agree with the solution, they can revise some configuration values. The approach records their preferences embedded in the revisions by tuning the weights of existing goals, so that subsequent adaptation results will be closer to the users' preferences. The experiments on a medium-sized simulated smart home system show that the approach is effective and scalable.

1 Introduction

Self-adaptability is an important feature of modern software-based systems. In adaptive systems, an adaptation agent monitors changes on a system or its environment, plans an appropriate configuration, and reconfigures the system accordingly [1–3]. Adaptation planning is guided by a set of policies, which specifies the desired system configuration under different contexts [4].

Adaptation policies are likely to be *incomplete* and *conflicting*: Under some particular context, either there may be multiple configurations that fit the goals, or no configuration can satisfy all the goals simultaneously. Such *imperfect* policies are practically unavoidable. Firstly, it is difficult for developers to eliminate all incompleteness and conflicts by enumerating every possible composition of the contexts to add extra policies. Secondly, the system may be constructed from existing components, each of which carries separately developed, and thus potentially conflicting, policies. At runtime, imperfect policies will result in multiple possible adaptation solutions. An adaptation agent has to choose one solution from them, but which one is the best may depend on who is using the system.

A promising improvement on self-adaptation is to tolerate imperfect policies, and do adaptation planning while considering end user preferences [1, 5,

* This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie)

6]. This paper takes such an approach, building the method over approaches based on models at runtime. [7, 2, 8]. By wrapping heterogeneous target systems as standard run-time models, we implement adaptations on top of model reading and writing, guided by model constraints in the form of OCL invariants. The challenges of this approach are threefold: 1) How could adaptation on runtime models and declarative model constraints be solved automatically; 2) How should user preferences be coded and utilized in adaptation planning; 3) What is an appropriate interface for end users to express their preferences?

The contributions of this paper can be summarized as follows.

- We design a partial evaluation semantics on OCL to automatically transform a run-time model into a constraint satisfaction problem (CSP) [9]. The variables of CSP are the context and configuration attributes, and constraints are from the current values of these attributes and the OCL invariants.
- We provide a novel approach to planning adaptations on CSP. We use *constraint diagnosing* to determine the optimal set of constraints to ignore, and then use *constraint solving* to assign new configuration values that satisfy the remaining constraints. User preference is reified as different weights of the constraints, and a constraint with higher weight will more probably be satisfied.
- We provide a straightforward way for end users to express their preference: After each round of adaptation, we allow users to directly redress the adaptation result, and in the background, we tune the weights of existing constraints according to the user's revision.

We evaluated the approach on a simulated smart home system. The adaptation significantly reduced the number of violated goals, and after a few rounds of preference tuning, the adaptation results became much closer to the simulated user preferences. The approach scales to medium-sized systems: An adaptation with 2000 constraints took 2 seconds on average. All implementation code and experiment results are available on-line [10].

The rest of the paper is structured as follows. Section 2 gives an overview of the approach, with a simplified running example. Sections 3 to 5 present the three technical contributions. Section 6 shows the experiments and results. Section 7 introduces related approaches, and Section 8 concludes the paper.

2 Approach Overview

2.1 Background and terminology

A *run-time model* is a model that presents a view of some aspect of an executing system [11]. In this paper we focus on structural run-time models [7] that present the structural composition of a system, and the attribute values of different compositional elements. Some of these attributes describe an observation of a system, such as room temperature, while some others describe ways to manipulate a system, such as the state of an electronically controllable switch. We

name these *contexts* and *configurations*, respectively. A run-time model is dynamically synchronized with the executing system, which means that the model state $s \in S$ (the elements and all their attribute values), at any time point, is the snapshot of the system at this time point, and transferring the state to s' (by giving new values to some configuration attributes) will cause the system to change accordingly. The set of all the possible model states S is defined by a meta-model. System adaptation based on a run-time model is a process to read the model state s, and then plan a new state s'.

We use model constraints on run-time models as the *adaptation policies* to guide the adaptation planning [1]. A model constraint is a function $cons : S \to \mathbb{B}$. For a model state $s \in S$, if $cons(s) = \top$ (we use $\top$ for *true* and $\bot$ for *false*) we say s satisfies *cons*. The objective of an adaptation is to make the model satisfy as many constraints as possible. From this perspective, the role played by model constraints in our approach conforms to the definition of *goal policies*, as they "directly specify desired system states" rather than "define how to achieve them" [5, 1]. In the rest of this paper, we call these model constraints *adaptation goals* in order to avoid ambiguity with the concept of constraints in CSP, as follows.

A *constraint satisfaction problem* (CSP) [9], or in particular a satisfiability modulo theory (SMT), is composed of a set of variables V and a set of first order logic constraints C over these variables. A constraint solver checks if there exists a labelling function $f : V \to D$ that assigns a value to each variable and these values satisfy all the constraints. If so, we say the CSP (V, C) is satisfiable, and the solver returns such an f. Some solvers divide constraints into *hard* and *weak* ones, i.e., $C = C_h \cup C_w$, and weak constraints can be ignored when necessary. For an unsatisfiable problem, the solver returns a sample of conflicting constraints from C_w. From the samples, we can construct a *diagnosis* $C_d \subseteq C_w$, such that $(V, C - C_d)$ is satisfiable. This process is called constraint diagnosing [12].

2.2 Motivating Example

We use a simple smart home system as a running example throughout this paper. Cheap but powerful sensors are now available, which collect a diversity of data from our living environment, and devices are frequently employed to make many household items electronically controllable. This enables and requires dynamic adaptation of an entire home when the environment changes, in order to improve living qualities and to save resources. At the same time, such a smart home is a highly personalized system. For the same context, different users may expect different adaptation effects.

Figure 1 describes a simplified domain model of smart home. The left part is a meta-model defining the types of system elements, the context information such as electricity price, time, room temperature, and the configuration points such as turning on or off the water heater. The heating system can work on different settings: 0 for off, 5 for fairly hot and 10 for very hot, etc. The right part of Figure 1 lists five adaptation goals, i.e., "when it is cold, the heatings should be at sufficient settings for comfort", "do not open window when the heating is on", "do open window when air quality is bad", "keep water heater on in the

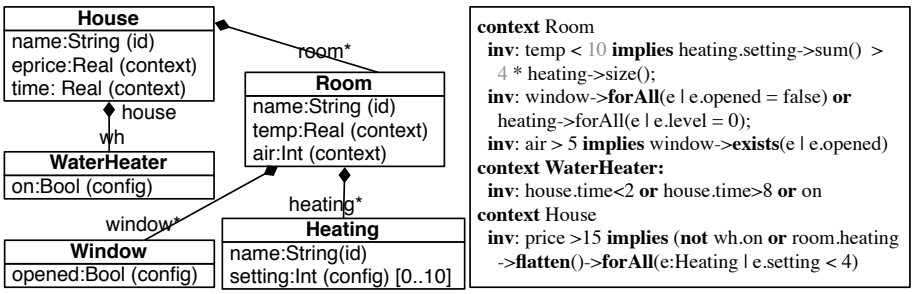

Fig. 1. Sample smart home meta-model and adaptation goals

early morning", and "when the electricity is expensive do not use water heater and strong heating together". The five sample goals are both *incomplete* and *conflicting*: The first does not point out a specific value for the heating settings. Alternatively, when it is cold and the air is bad, we can never satisfy the first three goals simultaneously.

Guided by such goals, there may not be a single perfect adaptation decision, and our solution is to take user preferences into account. For example, if we know the user prefers "heating 1 to work in setting 10", then we can choose setting 10 for this heating whenever it is one of the choices. For another example, if the user regards the third goal has a lower priority than others (he is more tolerable of smelly air), then when this goal conflicts with others, we sacrifice it first.

Since the smart home system targets end users who are probably without a computer science background, it is a burden for them to add new goals or tune the goal weights manually. Therefore, we provide a simplified interface: after each time of adaptation, the users can further *revise* the configuration by changing some of the attributes with the values they prefer. According to the revision, the approach generates goals or tunes the weights in the background.

2.3 The adaptation approach with user preference

This paper presents a dynamic adaptation approach guided by the domain model and user preferences. Figure 2 shows the approach architecture, where solid arrows indicate the main adaptation loop at runtime, and dashed arrows are the post-adaptation reference recording. The trident lines are the user intervention.

The system context and configuration are captured by a run-time model [7], such as the one shown in the left part of Figure 3. The construction and maintenance of run-time models are out of the scope of this paper, and some techniques can be found elsewhere [13, 14, 8]. Adaptation planning and preference tuning based on runtime models have the following three activities.

The approach first transforms the current run-time model into a CSP, such as the one shown on the right of Figure 3. It transforms each context or configuration attribute from each model element into a variable. The *hard constraints* are generated from configuration domains and current context values, as they

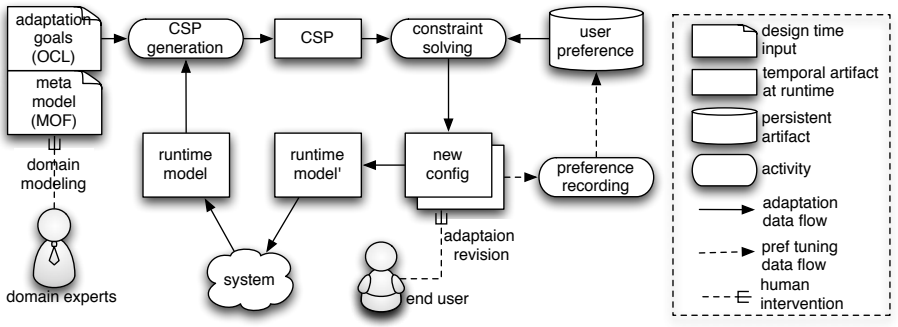

Fig. 2. Approach overview

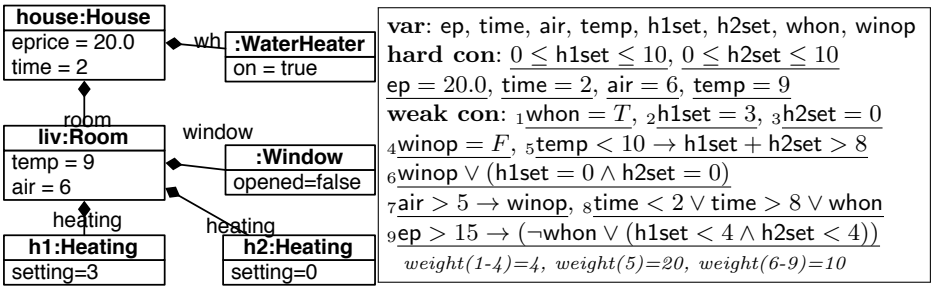

Fig. 3. Sample smart home runtime model and the generated CSP

cannot be violated after adaptation. The *weak constraints* are generated from configuration values and adaptation goals. The key technical idea here is a partial evaluation of the OCL language to identify the mapping from the attributes mentioned in OCL rules to the variables, which we will explain in Section 3.

Based on the generated CSP, the adaptation planning is to change the values of some of the configuration variables, so that the new configuration values, together with the current context, satisfy as many goals as possible. The planning begins by deciding which configuration variables to change, and which goals to ignore. This step boils down to finding an optimal *diagnosis* of the CSP. For example, from the CSP in Figure 3, we can find a diagnosis (2,7,9), such that if we change the value of `h1set`, we can find a solution(`h1set` = 9) to satisfy all the goals except 7 and 9. Since different diagnoses lead to different adaptation solutions, to help grade diagnoses (and the adaptation results), we assign each weak constraint a weight, and a constraint with bigger weight is more likely to be satisfied. The adaptation result is the one corresponding to the diagnosis with the minimal total weight. Under the weights shown in the bottom of Figure 3, diagnosis (2,7,9) has the minimal total weight 24. Section 4 presents our constraint diagnosing and solving approaches.

After each automated adaptation, if users change some of the configuration attributes, we record their preferences by tuning the weights of existing goals or generating new ones. Following our running example, if the user revises the

adaptation result by increasing `h1set` to 10, we will generate a new constraint `h1set` = 10. Now if in any case the adaptation engine needs to choose a value for `h1set`, it will first consider the value 10. As another example, if the user also opens the window, we will decrease the weights of 4 and 6, and increase the weight of 7 (as will be shown in Section 5). Next time under the same circumstance, we would find a different optimal diagnosis (2,4,6,9), and the window would be automatically opened.

3 Transforming a Run-Time Model to CSP

This section presents how we transform the run-time model and the OCL adaptation goals into a CSP. The **inputs** of the transformation are MOF meta-model and OCL invariants (Figure 1), and the current run-time model (Figure 3). The **output** CSP is in the form of variables and first order logic (FOL) constraints on them. The right part of Figure 3 illustrates the abstract and mathematical form of the CSP (the concrete form is in Z3Py [15]).

We generate a variable from each context or config attribute of each model element, i.e., $\mathsf{genvar} : M \times Elem \times Attr \rightharpoonup V$. For an element e from the current run-time model m, and an attribute $a \in e.Class.AllAttributes$ that is annotated as `context` or `config`, we get a variable $v = \mathsf{genvar}(m, e, a)$. From the model instance as shown in Figure 3, we generate 8 variables listed on the right. The constraints are generated from the domains of configuration attributes, the current context and configuration values, and the adaptation goals. Except for goals, the generation is straightforward, with self-explainable samples in Figure 3.

To transform the goals into FOL constraints, we replace the context and config attributes in the OCL invariants by the corresponding variables, resolve the static values in the run-time model, and maintain the operations between them. The challenge is that the OCL invariants are defined in the meta-model level, without concretely mentioning any model instances, whereas the FOL constraints are based on the variables that are generated from a particular model instance. We implement the transformation by defining a new *partial evaluation* [16] semantics on the OCL expressions, i.e., $[\![expr]\!]_{env} : M \rightarrow C$. The semantics on each expression *expr* is a function from a run-time model $m \in M$ to a constraint $c \in C$. Here `env` is an environment recording the mapping from OCL variables to values or model elements. As a simple example, if m is the model instance in Figure 3, then $[\![\texttt{self.temp < 10}]\!]_{\{\texttt{self}\mapsto\texttt{liv1}\}}(m) = t < 10$, where $t = \mathsf{genvar}(m, \texttt{liv1}, \texttt{temp})$.

Figure 4 lists an excerpt of the partial evaluation semantics on some typical forms of OCL expressions. For a data value in type of boolean, integer or real, we directly generate the value literal (1). For an OCL variable, we find its value from the environment and continue to evaluate this value (2). If the expression does not mention any context or configuration properties, we execute the OCL query to get its result (normally a value), and then evaluate the result (3). If the expression is to access a context or configuration property, we obtain the host model element and use its accessed attribute to locate the variable in CSP (4).

1. $[\![\mathsf{val}]\!]_c(m) = \texttt{literal}(\mathsf{val})$; 2. $[\![\mathsf{var}]\!]_c(m) = [\![c(\mathsf{var})]\!]_c(m)$;
3. $[\![\mathsf{expr}]\!]_c(m) = [\![\texttt{ocleval}(m, \mathsf{expr}, c)]\!]_c(m)$
if expr does not mention any context/config attributes
4. $[\![\mathsf{expr.attr}]\!]_c(m) = \mathsf{genvar}(m, \texttt{ocleval}(\mathsf{expr}, c), \mathsf{attr})$; *if attr is context/config*
5. $[\![\mathsf{expr}_1 + \mathsf{expr}_2]\!]_c(m) = [\![\mathsf{expr}_1]\!]_c(m) + [\![\mathsf{expr}_2]\!]_c(\mathrm{m})$;
6. $[\![\mathsf{expr}_1$ **and** $\mathsf{expr}_2]\!]_c(m) = \mathrm{And}([\![\mathsf{expr}_1]\!]_c(m), [\![\mathsf{expr}_2]\!]_c(m))$;
7. $[\![\mathsf{expr}$**->sum**$]\!](m) = [\![v_1]\!]_c(m) + ... + [\![v_n]\!]_c(m), v_1...v_i \in \texttt{ocleval}(m, \mathsf{expr}, c)$;
8. $[\![\mathsf{expr}_1$**->forAll**$(\mathsf{e}|\mathsf{expr}_2)]\!]_c(m) =$
$\mathrm{And}([\![\mathsf{expr}_2]\!]_{c\cup\{e\mapsto v_i\}}(m),...)$, $v_i \in \texttt{ocleval}(m, \mathsf{expr}_1, c)$;
9. $[\![$**let** $\mathsf{e} = \mathsf{expr}_1$**in** $\mathsf{expr}_2]\!]_c(m) = [\![\mathsf{expr}_2]\!]_{c\cup\{\mathsf{e}\mapsto\texttt{ocleval}(\mathsf{expr}_1, c)\}}(m)$;

Fig. 4. The partial evaluation semantics on OCL to generate constraints

For the mathematical and logical OCL operations, we generate the corresponding FOL operation (5 and 6), following the Z3Py format (e.g., it uses `And(a,b)` for conjunction). For an operation on collection, we obtain the host collection first, and then combine the partial evaluation of each collection item (7, 8). For `let` or iteration expressions where new variables are introduced (8, 9), we resolve the variable value first and put it to the environment before evaluating the sub expressions. Using this semantics, we transform the OCL invariants by traversing all the model elements in the current run-time model.

4 Adaptation planning based on CSP

This section presents the adaptation planning based on a generated CSP $(V, C_h \cup C_w)$. If the CSP is satisfiable we do not need to do anything for adaptation. Otherwise, we plan the adaptation by *constraint diagnosing* and *constraint solving*.

In order to grade the diagnoses, we attach each weak constraint a weight, $\mathsf{weight} : C_w \to \mathbb{N}$, and the target of adaptation planning is to find the diagnosis with the minimal total weight. From C_d, we perform constraint solving on $(V, C_h \cup C_w - C_d)$, and obtain a new configuration f which is a mapping from each config variable to a value, where $f \vdash (C_h \cup C_w - C_d)$.

Our algorithm to search for the optimal diagnosis is inspired by the work of Reiter [17] and Greiner et al. [12] on non-weighted CSPs, which is essentially a breadth-first searching for minimal hitting sets, each of which covers all the sample conflicting sets returned by the solvers. With the help of the constraint weights, we leverage a dynamic programming approach similar to the Dijkstra shortest path algorithm.

We illustrate the basic idea of our algorithm using the sample CSP in Figure 3, and its execution process is shown in Figure 5. We first ask the solver for an arbitrary sample set of conflicting constraints, and it returns {4,7}, meaning that the closed window conflicts with the "should open window" goal. To make the CSP satisfiable, we must open the window or ignore the goal, and thus we make (4) and (7) as two candidate diagnoses, and pick the one with the lowest

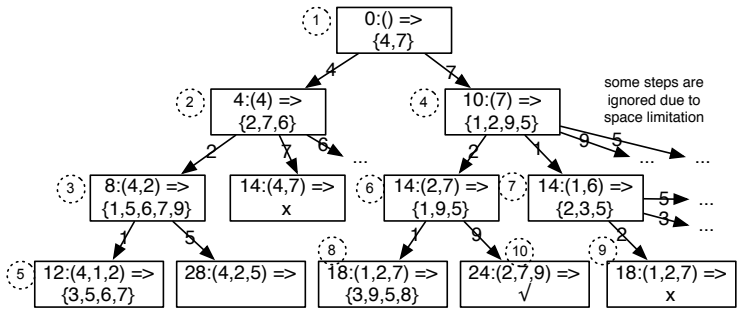

Fig. 5. Sample diagnosing process

total weight to check (i.e., (4) weights 4). Here *checking a candidate* means that we remove the constraints in this candidate from the original CSP, and ask the solver for a new sample conflict set. By checking (4), we get a sample set {2,7,6}, that means if we remove 4, there is still a conflict between 2, 7 and 6 ("bad air, open window" conflicts with "heating on, close window"). So we extend the candidate by each of the conflicting constraints, and get three new candidates. Now we have four unchecked candidates, i.e., (4,2), (4,7), (4,6), and (7). We also pick the one with the lowest total weight (i.e., (4,2)) to check and expand. After no candidates weighted 10 left unchecked, we check (7) and get {1,2,9,5}, which leads to four new candidates. After having all the candidates under 24 checked, we check (2,7,9), and the solver returns "satisfiable". Its intuitive meaning is to change the first heating, ignore the bad air and high electricity price.

Algorithm 1 lists the steps of the diagnosing algorithm. We maintain a set `cands` of all the unchecked candidates, and put an empty set as the initial diagnosis (Line 1). The main part of the algorithm is a loop (Lines 3-17). Each time, we take out the candidate diagnosis `curr` which has the minimal total weight (Line 4). Then we check whether removing `curr` will make the CSP satisfiable (Line 10, we skip Lines 5-10 first as they are related to optimization). If the `check` succeeds, we return `curr` as the final diagnosis, and terminate the algorithm (line 10), otherwise, we ask the solver to provide a new sample(Line 12). If the solver cannot find any (this only happens when the hard constraints are conflicting), we terminate the algorithm without a solution (Line 13). After having the new sample `newsamp`, we take each constraint c from it (Line 15), expand the current candidate by adding c into it and push it into the pool (16).

We employ the following optimization. Firstly, since the diagnosis should have an intersection with every conflicting sample [17], we cache all the samples returned by the solver (Line 14). When dealing with any candidate, we first check if it covers all the cached samples (Line 9), and if not, we use one uncovered sample from the cache to extend the candidate instead of bothering the solvers to produce one. The second goal is to accelerate the set operations. We implement `cands` as a heap queue, so that we can push an item or pop the smallest one in $O(\log n)$. Since it is hard to filter identical items in a heap queue, we maintain

Algorithm 1: Weight-based constraint diagnosing

In: Variables V, hard constraints C_h, weak constraints C_w, and weight
Out: A diagnosis $C_d \subseteq C_w$, with the lowest total cost

```
1  cands←{φ} ;
2  cache←{}, visited←{}, lastweight← − 1 ;
3  while cands is not empty do
4      curr ← pop (cands) where currweight← ∑_{c∈curr} weight(c) is minimal ;
5      if lastweight = currweight then
6          if curr ∈ visited then continue;
7          else visited←visited ∪ {curr}
8      else lastweight←currweight, visited←{} ;
9      if ∃(s ∈ cache)[s ∩ curr = φ] then newsamp←s ;
10     else if satis(V, C_h, C_w − curr) = T then return curr as C_d;
11     else
12         newsamp ← sample(V, C_h, C_w − curr) ;
13         if newsamp = φ then throw 'conflicts in hard constraints';
14         else cache←cache ∪ {newsamp} ;
15     foreach c ∈ newsamp do
16         newcand←curr ∪ {c}; push(cands, NewCand) ;
```

a list of recently visited candidates with the same particular total weight, and use it to check if the current candidate has been visited (Lines 5-7).

5 End User Preference Recording

If users revise an adaptation result, we reify their preferences by tuning the weights of existing constraints or generating new ones, so that the subsequent adaptation results will be closer to the one that users preferred.

In order to tune the CSP, the first task is to identify the user's *preferred diagnosis* corresponding to the revised configuration they provide. Formally speaking, for a revised configuration f', the preferred diagnosis $C'_d \subseteq C_w$ holds that $f' \vdash (C_h \cup C_w - C'_d)$. From the configuration f', it is straightforward to reversely derive the diagnosis C'_d: Just find all the original weak constraints that cannot be satisfied by the current configuration f', i.e., $C'_d = \{c \in C_w | \neg(f' \vdash c)\}$.

We handle the weight tuning separately for the two different containment relationships between C_d and C'_d, as shown in Figure 6.

Firstly, if $\neg(C_d \subseteq C'_d)$, as shown in Figure 6(a), we have three subsets namely I: $C_d - C'_d$, II: $C'_d - C_d$ and III: $C_d \cap C'_d$. III can be empty, but I and II can not[1]. We increase the weight of each constraint in I, and decrease II. In particular, $\mathsf{weight}'(c \in \mathrm{I}) = \mathsf{weight}(c) \times (\sum_{i \in \mathrm{II}} \mathsf{weight}(i) / \sum_{j \in \mathrm{I}} \mathsf{weight}(j))$,

[1] $C'_d \subseteq C_d$ never happens, otherwise C_d cannot be a minimal diagnosis

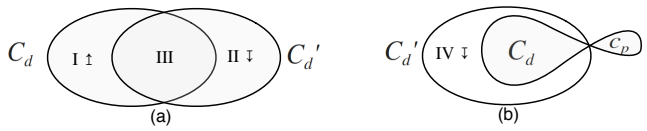

Fig. 6. Two different conditions for constraint weight tuning

$\mathsf{weight}'(c \in \text{II}) = \mathsf{weight}(c) \times (\sum_{i \in \text{I}} \mathsf{weight}(i) / \sum_{j \in \text{II}} \mathsf{weight}(j))$. In this way, we change all the weights in proportion, and switch the total weight of C_d and C'_d. The intuitive meaning of this tuning can be illustrated by the following example. The adaptation result in the last section $h1set = 9$ corresponds to $C_d = \{2, 7, 9\}$. Suppose the user further modify the configuration by opening the window ($winop = T$), and this new f' corresponds to $C'_d = \{2, 4, 6, 9\}$. Now we have I= $\{7\}$ and II= $\{4, 6\}$. We get $\mathsf{weight}'(7) = 10 * (4 + 10)/10 = 14$, which means that the user is reluctant to break constraint 7 (bad air `->` open window), and we get $\mathsf{weight}'(4) = 2$ and $\mathsf{weight}'(6) = 6$, indicating he does not care about 4 (keep the window's status) and 6 (do not open window when heating).

Secondly, if $C_d \subseteq C'_d$, as shown in Figure 6(b). We also decrease the weights in IV=$C'_d - C_d$. However, since the weight is not negative, we cannot make the total weight of C'_d less than that of C_d. Therefore, we introduce a set of new constraints C_p: for each variable $v \in V$, if user modified it with a new value d, then $v = d$ is a constraint in C_p. Now under the new CSP $(V, C_d, C_w \cup C_p)$, the original configuration f corresponds to diagnosis $C_d \cup C_p$, whereas the user's preferred configuration f' still corresponds to C'_d. We make $\mathsf{weight}'(c \in \text{IV}) = \mathsf{weight}(c)/2$, and $\mathsf{weight}'(c \in C_p) = \max(\sum_{i \in IV} \mathsf{weight}(i)/|C_p|, \texttt{default})$. For example, from the same f and $C_d = \{2, 7, 9\}$, if the users modify the result by further set $h2set = 5$, then this new f' corresponds to $C'_d = \{2, 3, 7, 9\}$. So we have IV=$\{3\}$, $C_p = \{10 : \texttt{h2set} = 5\}$. The intuitive meaning is that the user would like to turn `h2` to setting 5 when possible. This condition also covers $C_d = C'_d$, where we only generate new constraints, without tuning any weights.

6 Evaluation

The implementation of the whole approach has two parts. We implement the CSP generation engine by Xtend [18], reusing the Eclipse OCL library for the parsing of OCL texts. We choose Microsoft Z3 [15] as the constraint solver, and implement the adaptation planning and preference tuning in Python.

We use a simulated smart home system to evaluate the effect and performance of the approach. The target system simulates a typical home similar to the example described in Section 2, but much more complex. The setting of this simulated system is based on existing smart home projects, especially the "Adaptive House" from University of Colorado [19], and the household level smart grid research from our own group [20]. The table in Figure 7 summarizes

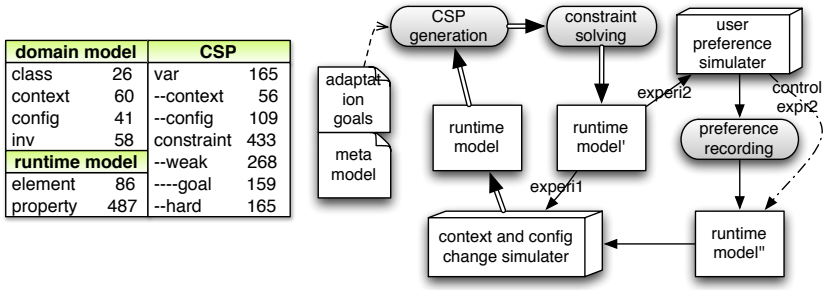

Fig. 7. Experiment setting

the sample system. The main run-time model corresponds to a fictitious 10-room house with full equipments. From this run-time model, we generate a CSP that contains 165 variables and 433 constraints.

All the implementation source code, the experiment artefacts, and the results mentioned in this section can be found in our GitHub repository [10].

6.1 Effectiveness

On this target system, we perform two experiments to answer the following two questions: 1) Does the adaptation make the system more consistent with adaptation goals? 2) Will the adaptation results more closely match users' preferences, after users revise the adaptation outcome over a few iterations.

The first experiment follows the small central circle in Figure 7. We implement "change simulator" to randomly modify the attributes in the run-time model to simulate the system evolution. We perform adaptation planning on model with state s, and get a new model state s' as the adaptation result. After that, s' is fed to the simulator, which randomly modifies the attributes again and starts the next round of adaptation. For each round of adaptation, we care about how s' is *improved* compared to s, in terms of what proportion of the goals violated by s are satisfied by s'.

Figure 8(a) shows the results. We run the adaptation 100 times, each of which is represented by a vertical arrow. The start and end points of an arrow corresponds to the numbers of violated goals before and after an adaptation, respectively. The x axis describes how many configuration variables are changed by the adaptation (The adaptations with 0 changes are not displayed). We can see that in most cases, the adaptation has a significant improvement, reducing the number of violated goals from 10-20 to 1-4, and these improvements are mostly achieved by modifying 6-12 configuration values.

The second experiment evaluates the effect of preference tuning. We implement another simulator to act as an imitated user, and embedded 11 preference rules in the simulator, in the form of `condition->config=value` ("under a special `condition`, I prefer the value of a `config` variable to be equal to the a

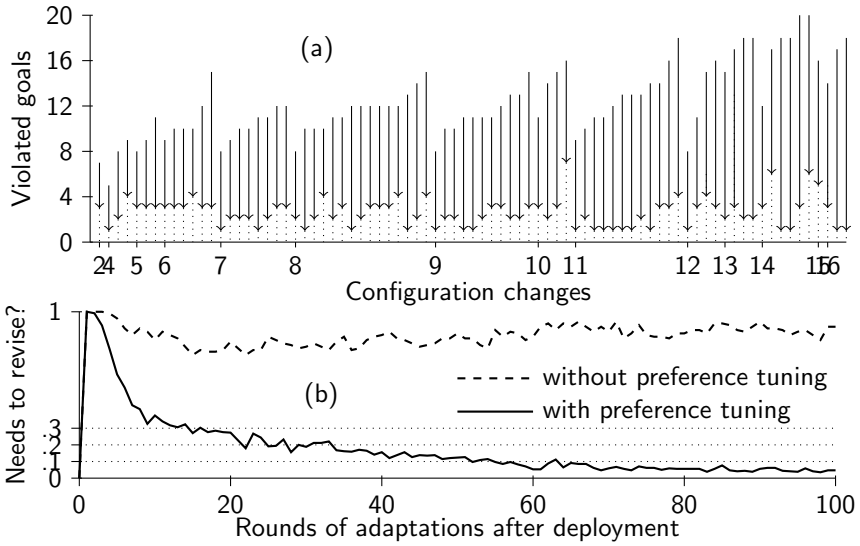

Fig. 8. Effects of adaptation and user preference tuning

specific `value`"). Such a rule is either a refinement to an adaptation goal (e.g., "if a heating setting is greater than 0, then it should be 10"), or an insistence to a particular goal (e.g., "whenever the air condition is bad, the window should be opened"). As shown by the bigger circle in Figure 7, after each adaptation, the user simulator evaluates the results. If any preference rule is violated, it picks one and only one from them, changes the configuration value according to the `config=value` part, yielding a new model s''. After that, the system simulator will randomly change context and configuration on s'', and start the next round of adaptation. Ideally, after more revisions, the automated adaptation result s' will be less probable to violate any preference.

Figure 8(b) illustrates the results. We run the experiment 1000 times, each time with 100 rounds of the adaptation/tuning loop as shown in Figure 7. The solid line illustrates that for the xth round of adaptation after initialization, the adaptation output has y possibility to violate one or more preference rules (i.e., 1000*y times of violation is observed in the 1000 times of experiments). We can see that the possibility of preference violation goes down very quickly. After only 10 rounds (each round with at most one refinement on one configuration!), there is only 30% possibility to violate any preference, and after about 60 rounds, it is less than 10%. As a comparison, we run another 1000 times of experiments bypassing the preference tuning, and the possibility of preference violation is shown by the dashed line. The experiment shows that the preference tuning has a good effect even on such an exaggerated situation (almost every time the adaptation will violate a preference rule).

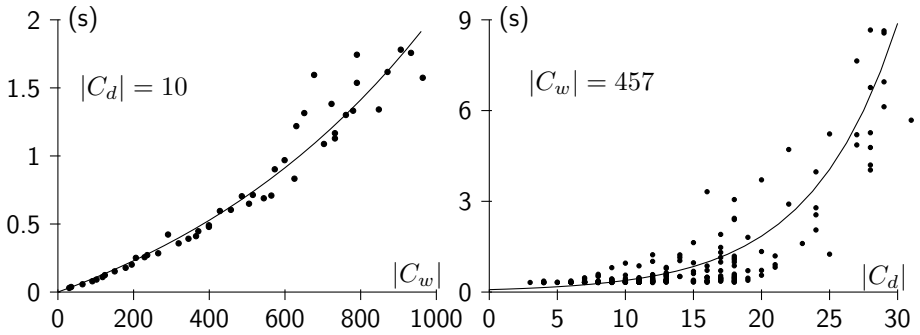

Fig. 9. Experiment results on performance

6.2 Scalability

The scalability of the whole approach mainly depends on the execution time of the adaptation planning step, because the other two only have a linear time complexity with the scale of run-time models. There are two main factors influencing the performance of this step, i.e., the size of the weak constraints $|C_w|$, and the size of the resulting diagnosis $|C_d|$. We show the influence of these factors respectively. The experiment processes are the same as the `experi1` in Figure 7. All the experiments are performed on a MacBook Pro laptop with Intel i5 2.0GHz CPU and 4GB memory. The software environment includes MacOS X 10.5, Microsoft Z3, and Python 2.7.3. The weights of constraints from current configurations are randomly chosen between 200 to 300, and for goals, the range is 2000 to 3000.

To evaluate the influence of $|C_w|$, we keep adding elements to the runtime model, and this leads $|C_w|$ growing from 30 to 957. For each run-time model, we run the experiment for 100 times, and select the adaptation whose diagnosis size equals to 10. The left part of Figure 9 shows the average time on each run-time model. According to the approximate fitting curve, the time still grows exponentially with the size of constraints, but in a quite flat way.

To evaluate the influence of $|C_d|$, we choose one run-time model with $|C_w| = 457$, and control $|C_d|$ by making the simulator change more context and configurations variables, sometimes with extreme values. We run 1000 rounds of adaptations, and record the size of diagnosis and the execution time of each round in the right part of Figure 9: The execution time ascends quite fast with the increase of $|C_d|$, reaching a maximal 9 seconds when $|C_d| = 29$, though 96% of the adaptations finished within 2 seconds.

6.3 Threats and Discussion

All the experiments are performed on the same domain model, and similar run-time models with different sizes. Therefore the effectiveness and performance can be affected by some specific features of this experiment system. To alleviate this threat, we defined the run-time model and the goals independently before the

experiments. The effectiveness of preference tuning strongly depends on the type of preferences. A "refinement preference" rule can be only violated once on the same element, whereas an "insistence to a goal" rule may be violated multiple times. We designed the imitated preference rules with a balance of both types. Note that if the preference rules themselves have conflicts, the preference tuning is not convergent. We avoid such conflicts, imitating a "firm-minded" user who does not change his mind on preference. The size of diagnosis currently has a ceiling of 30. A well designed domain model with fewer conflicting goals will significantly reduce the diagnosis size, and therefore increase the adaptation performance.

7 Related Work

Models at runtime are widely used to support dynamic system adaptation. Garlan et al. [2] and Sicard et al. [14] execute action policies on run-time architecture models to achieve self-optimization and self-repair, respectively. Morin et al. support dynamic adaptation by executing the "aspect-oriented rules" defined on run-time models [8, 21]. We follow the same idea, executing adaptation policies on a basic type of run-time models that present only the structural system aspects [7]. Our innovation is to tolerate conflicting policies, with the consideration of end user preference. This paper is also an attempt of a novel way to leverage models at runtime for adaptation: Based on the fact that run-time models are formal descriptions of run-time system states, we utilize a mathematical tool, constraint solving, to achieve automated adaptation planning guided by declarative OCL constraints defined on the models.

Salehie and Tahvildari [1] regard the consideration of user preference as an essential aspect of self-adaptive software, but also note that the research in this direction is still in an early stage. Maximilien et al. [6] utilize a set of user preference policies in addition to the business policies, to improve the automated service selection. Kephart [5] proposes to support user preference by a *flexible interpretation of adaptation policies*. Our approach follows this direction by using tunable weighted goal policies. We support a straightforward interface for end users, conforming to Russell et al.'s principle on user experience of autonomic computing, i.e., only showing users the understandable actions [22].

This approach has a similar motivation with goal-based adaptation approaches [23–25], i.e., to increase the abstraction level of adaptation, but the two branches of work address different concerns. Those approaches focus on the modelling of adaptive systems, from a requirement perspective, and achieve this by adopting and extending a requirement modelling concept, the *goals*. To implement the adaptive systems, they link goals to lower-level modelling elements, such as tasks [25] or operations [23], or statically calculate the potential target systems for different contexts [24, 26]. Our approach is focused on how to plan the adaptation at runtime from a declarative specification about the desired system. As a first attempt, we choose model constraints as such a specification. We also name them *goals* for short as they conform to the definition of "goal policy" from

adaptive system literature, but they do not have as strong expression power as requirement goals, such as the goal decomposition hierarchy. From this point of view, the two branches are complementary: We will investigate how to utilize requirement goal-based models as an input to our approach, and by doing this, we can provide a possible way to execute the requirement goal models at runtime without mapping them to imperative specifications.

Adaptation planning on a run-time model is similar to inconsistency fixing on a static model under editing. Recent approaches seek the automated inference of fixing updates by designing extra fixing semantics on the constraint languages [27], or analysing the relation between previous editing and fixing actions with the constraint rules [28]. Xiong et al. use constraint solving to construct the recommended fix ranges for configuration models at runtime [29]. Instead of a range, adaptation requires a specific value for each configuration item, and we choose such a value with the help of user preferences.

Constraint solving has been used by others for self-adaptation. Sun et al. use constraint solving to verify the role based access control policies [30]. White et al. use constraint solving to guide the system configuration on feature models [31]. Sawyer et al. [26] do constraint solving on the requirement-level goals to solve out proper configurations for all the possible contexts, and use the result to guide run-time adaptation. Instead of analysis in advance, we use constraint solvers at runtime, while the target model is still changing. Neema et al. [32] present a similar constraint guided adaptation framework but our work is focused more on the technical solution about how to solve constraints. This work is related to the generation of CSP from class diagrams [33, 34] or OCL constraints [35]. However, we do not perform verification merely on meta-models, but also use a model instance as a seed, based on a new partial evaluation mechanism.

8 Conclusion and Future Work

This paper reports our initial attempt towards automated system adaptation with the consideration of end user preferences. We transform run-time models into a CSP, and perform constraint diagnosing and solving to plan the new system configuration. If users revise the automated calculated configurations, we record their preferences by tuning the constraint weights, and subsequent adaptations will then yield results closer to the users preferences.

Our future plan is to apply the approach to different domains, and evaluate the experience from real users. Instead of supporting only the adaptation of attribute values, we will investigate how to transform the more complicated model changes into CSP, such as changing references between elements. In order to improve the scalability, we are now investigating the usage of AI search techniques on constraint diagnosis. Another important direction is to enhance the interface of this work to both developers and end users: For developers, we will investigate how to derive model constraints from higher-level models, such as requirement goals; For end users, we will provide a more interactive graphical user interface for them to revise the adaptation results.

References

1. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems (TAAS) **4**(2) (2009) 14
2. Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer **37**(10) (2004) 46–54
3. Cheng, B.H., De Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., et al.: Software engineering for self-adaptive systems: A research roadmap. Springer (2009)
4. Kephart, J., Walsh, W.: An artificial intelligence perspective on autonomic computing policies. In: IEEE International Workshop on Policies for Distributed Systems and Networks, IEEE (2004) 3–12
5. Kephart, J.: Research challenges of autonomic computing. In: ICSE, IEEE (2005) 15–22
6. Maximilien, E., Singh, M.: Toward autonomic web services trust and selection. In: Proceedings of the 2nd international conference on Service Oriented Computing, ACM (2004) 212–221
7. Blair, G., Bencomo, N., France, R.: Models@ run. time. Computer **42**(10) (2009) 22–27
8. Morin, B., Barais, O., Nain, G., Jezequel, J.: Taming dynamically adaptive systems using models and aspects. In: ICSE, IEEE Computer Society (2009) 122–132
9. Kumar, V.: Algorithms for constraint-satisfaction problems: A survey. AI magazine **13**(1) (1992) 32
10. Song, H.: All the source code, experiment resource, and results mentioned in this paper, hosted by github. `https://github.com/songhui/cspadapt`
11. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: 2007 Future of Software Engineering, IEEE Computer Society (2007) 37–54
12. Greiner, R., Smith, B., Wilkerson, R.: A correction to the algorithm in reiter's theory of diagnosis. Artificial Intelligence **41**(1) (1989) 79–88
13. Song, H., Xiong, Y., Chauvel, F., Huang, G., Hu, Z., Mei, H.: Generating synchronization engines between running systems and their model-based views. Models in Software Engineering (2010) 140–154
14. Sicard, S., Boyer, F., De Palma, N.: Using components for architecture-based management: the self-repair case. In: ICSE, ACM (2008) 101–110
15. Microsoft Research: Z3: a high-performance theorem prover. `http://z3.codeplex.com`
16. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice Hall (New York) (1993)
17. Reiter, R.: A theory of diagnosis from first principles. Artificial intelligence **32**(1) (1987) 57–95
18. Xtend: a statically-typed programming language which compiles to comprehensible java source code. `http://www.eclipse.org/xtend/`
19. Mozer, M.: The adaptive house. `http://www.cs.colorado.edu/~mozer/nnh/`
20. Galvan, E., Harris, C., Dusparic, I., Clarke, S., Cahill, V.: Reducing electricity costs in a dynamic pricing environment. In: IEEE SmartGridComm, IEEE 169–174
21. Morin, B., Mouelhi, T., Fleurey, F., Le Traon, Y., Barais, O., Jézéquel, J.: Security-driven model-based dynamic adaptation. In: ASE, ACM (2010) 205–214

22. Russell, D., Maglio, P., Dordick, R., Neti, C.: Dealing with ghosts: Managing the user experience of autonomic computing. IBM Systems Journal **42**(1) (2003) 177–188
23. Baresi, L., Pasquale, L., Spoletini, P.: Fuzzy goals for requirements-driven adaptation. In: RE, IEEE (2010) 125–134
24. Cheng, B.H., Sawyer, P., Bencomo, N., Whittle, J.: A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In: MODELS. Springer (2009) 468–483
25. Dalpiaz, F., Giorgini, P., Mylopoulos, J.: An architecture for requirements-driven self-reconfiguration. In: Advanced Information Systems Engineering, Springer (2009) 246–260
26. Sawyer, P., Mazo, R., Diaz, D., Salinesi, C., Hughes, D.: Using constraint programming to manage configurations in self-adaptive systems. Computer **45**(10) (2012) 56–63
27. Nentwich, C., Emmerich, W., Finkelstein, A.: Consistency management with repair actions. In: ICSE, IEEE (2003) 455–464
28. Egyed, A., Letier, E., Finkelstein, A.: Generating and evaluating choices for fixing inconsistencies in uml design models. In: ASE, IEEE (2008) 99–108
29. Xiong, Y., Hubaux, A., She, S., Czarnecki, K.: Generating range fixes for software configuration. In: ICSE, IEEE (2012) 58–68
30. Sun, W., France, R., Ray, I.: Rigorous analysis of uml access control policy models. In: Policies for Distributed Systems and Networks (POLICY), 2011 IEEE International Symposium on, IEEE (2011) 9–16
31. White, J., Dougherty, B., Schmidt, D., Benavides, D.: Automated reasoning for multi-step feature model configuration problems. In: Proceedings of the 13th International Software Product Line Conference, Carnegie Mellon University (2009) 11–20
32. Neema, S., Ledeczi, A.: Constraint-guided self-adaptation. Self-Adaptive Software: Applications (2003) 325–327
33. Maoz, S., Ringert, J., Rumpe, B.: CD2Alloy: Class diagrams analysis using alloy revisited. MODELS (2011) 592–607
34. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: Software Testing Verification and Validation Workshop, IEEE (2008) 73–80
35. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a tool for the formal verification of uml/ocl models using constraint programming. In: ASE, ACM (2007) 547–548