ISSN 1884-0760

GRACE TECHNICAL REPORTS

A Novel Approach to Goal-oriented Adaptation with View-based Rules

Tianqi Zhao Tao Zan Haiyan Zhao
Zhenjiang Hu Zhi Jin

GRACE-TR 2016–01 February 2016

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

The GRACE technical reports are published as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, notwithstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

A Novel Approach to Goal-oriented Adaptation with View-based Rules

Tianqi Zhao* Tao Zan+ Haiyan Zhao*
Zhenjiang Hu*+ Zhi Jin*
*Institute of Software, School of EECS, Peking University
Beijing, 100871, China
zhaotq12, zhhy, zhijing@sei.pku.edu.cn
+National Institute of Informatics
The Graduate University for Advanced Studies
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430
zantao, hu@nii.ac.jp {hu,kato}@nii.ac.jp

June 12th, 2009

Abstract

Rule-based adaption provides a powerful mechanism to program adaptable software, where rules specify adaptation logic of what particular action should be performed to react to monitored events. It has advantages of readability and elegance of each individual rule, the efficiency of plan process, and the ease of rule modification. However, adaptation rules in the existing approaches are not structured well, which makes it difficult to deal with efficient conflict resolution, to be seamlessly combined with user's goal and requirements, and to evolve dynamically. In this paper, we propose a novel idea of νRule for structuring adaptation rules. The structured adaptation rules are expressive enough for programming intended adaptation logic, and their well-behavedness that is automatically checked at the design time can guarantee that they will not lead to any conflict at runtime. In addition, we show that νRule provides a flexible mechanism for users to customize and evolve the adaptation systems. We have designed and implemented a new view-based adaptation framework for supporting construction of adaptive systems, based on νRule and the feature modeling technique, and successfully apply it to realize a nontrivial smart home system.

1 Introduction

Rule-based adaptation [1, 2, 3] provides a powerful mechanism to develop self-adaptive systems, enabling systems to modify their behavior, reconfigure their structure, and evolve over time reacting to changes in the operating context [4]. In a

rule-based adaptation system, a set of adaptation rules are used to specify adaptation logic of what particular action should be performed to react to monitored events.

Typically, an adaptation rule takes the form of "$condition \Rightarrow action$" where *condition* specifies the trigger of the rule, which is often fired as a result of a set of monitoring operations, and *action* specifies an operation sequence to perform in response to the trigger. For instance, in a smart room system (as will be explained more in Section 2), we may have the following rule

$$\begin{aligned} & Light.Power = \mathit{off} \ \wedge \ Time = daytime \\ & \quad \Rightarrow Blind.State := open; \ Window.State := open \end{aligned}$$

which declares that if the light is power off in daytime, then open the blind and the window. Obviously, rule-based adaptation has advantages of readability and elegance of each individual rule, the efficiency of plan process, and the ease of rule modification.

In spite of these advantages, adaptation rules pay attention only to local transformation, which makes it difficult to satisfy user global goal that expresses the purpose of the developed system. Moreover, like the operating context, user goal setting may change dynamically, and good self-adaptive systems should be flexible to adapt accordingly without intervention from technicians. However, the adaptation rules are static so that the adaptation logic defined by the rules cannot change at runtime, which prevents it from being adaptable to dynamic goal change.

On the other hand, the goal based and utility function based approaches [5, 6] provide a solution to make adaptation plans that can match with changed goal. They normally reduce the dynamic adaptation as a linear programming problem and leave the system to reason on the actions required to achieve high-level goals or optimize utility functions. While, despite a greater possibility to find an optimized solution at runtime, they always encounter large resource consumption and suffer from high execution cost.

In this paper, we propose a novel approach to enriching rule-based adaptation with goals by *structuring* adaptation rules with *invariant*. The key idea is to (1) refine the rule by splitting the condition part into two as $view \wedge condition \Rightarrow action$, where $view$ denotes an *invariant* that will be preserved after the action, and (2) impose the following semantics to the rule: "Under $view$, if $condition$ is satisfied, do $action$ while keeping the $view$."

The important point is the use of invariant for capturing the view state that a rule should maintain in adaptation. Since a user goal is usually realized by a (proper) state [5], it is this view state that enables us to associate rules with goals. To make the $view$ stand out in the rule, we shall write the rule as

$$view \vdash condition \Rightarrow action$$

and call it view-based rule (νRule for short) in this paper.

The main technical contributions of this paper can be summarized as follows.

- We present a novel concept of νRule , where an invariant view is introduced for structuring adaptation rules and relating them with goals (Section 3.2). As a refinement of the traditional adaptation rules, νRule s are expressive enough for programming intended adaptation logic. On the other hand, thanks to more refined structure and added invariant semantics, νRule s serve effectively as basic units for construction of various powerful and well-behaved adaptation rule systems at both design and run times.

- We propose a new view-based adaptation framework for supporting construction of adaptive systems based on νRule and the goal-oriented modeling technique (Section **??**). It seamlessly integrates the rule-based planning with the goal-based planing, gaining the advantages of both traditional rule-based and goal-based adaptation approaches. Accordingly, the run-time adaptation can efficiently response to the changes in both the environment and the user goal setting.

- We have implemented the framework[1] with two newly developed algorithms (Section **??**). One is for goal-based planning where we introduce *property* to bridge the gap between goals and features and divide the optimization procedure into two steps: goal-based reasoning in section 3.4.1 and strategy derivation in section 3.4.2. The other is for dynamic rule generation, where by use of the good connection between the goal-related feature view and the νRule structure, a well-behaved goal-related νRule set is dynamically derived from the νRules in the static knowledge base.

We have applied the framework to design a smart room system. Our experimental results (Section **??**) show that (1) our approach reaches a much higher degree of goal satisfaction than the traditional rule-based adaptation approach; and (2) our approach scales and works more efficiently than the traditional goal-based approach.

2 Running Example: Smart Rooms

Let us imagine such a typical scenario: developers want to construct a smart home for its residents as shown by the goal model [7] in Figure 1. A smart home system is usually designed under three main concerns: it should reduce energy consumption, provide a high level of resident comfort and ensure home security. In the literature of smart home [8], thermal comfort, visual comfort and good air quality are regarded as three basic factors that determine the quality of life in buildings. Besides, different smart home system take some more factors into account, for example, acoustic comfort. These factors can be treated as goals that adaptive system should satisfy.

An example of dynamic adaptation to meet the goals *suitable lighting intensity* and *suitable screen brightness* are as follows. A computer with adaptive capacity can adjust itself to indoor lighting intensity. When the residents modify computer

[1] The system is available at http://www.prg.nii.ac.jp/members/stefanzan/viewrule.html.

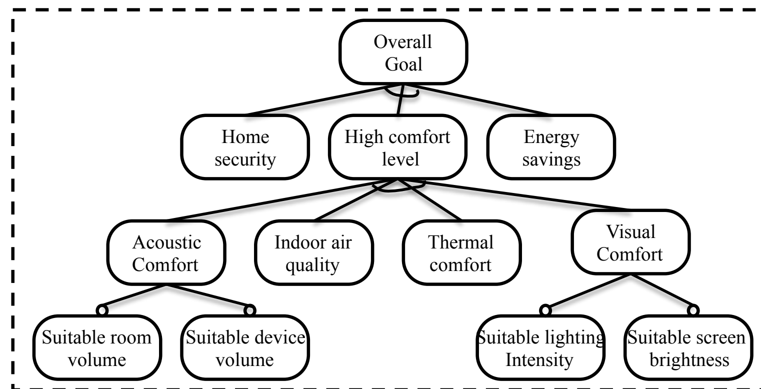

Figure 1: A Goal Model for Smart Home

screen brightness, this modification should be kept to make the user feel that he has full control of the room. At the same time, if there comes a request of *suitable light intensity* for the room, it should give several possible resolutions to the request, such as, turning on the light, opening the blind or both of them.

The adaptation mechanism in a smart home system should not be immutable, rather, the mechanism itself should also be adaptive to user goals, priority of goals, and user preferences. This is because that different residents might pay their attention to different goals according to their interests. Generally, a self-adaptive system could well adapt itself to dynamic changing environment without human intervention. However, for a smart home, its resident may need to have a full control of all the home states, and have the permission to add new devices and new rules into the adaptation engine.

It is, however, not easy for the developers to build such a smart home system that meets all these requirements. To set the smart room developers free from this tough task, it will be helpful to have a new framework to facilitate them to construct such kinds of smart home system in an easy way like following: 1) formalizing the smart home context, including the physical environment, the devices and their reconfigurable parameters, by the feature modeling technique; 2) specifying adaptation logic by a set of adaptation rules that associate with user goals. With these two kinds of information, the new framework will generate automatically an adaptation engine for the smart home system.

The following characteristics are desired for the new framework:

• *Free from conflicts.* As long as the adaptive rules provided by the developer meet the standards of our framework, possible conflicts of these adaptive rules would be detected statically and there will be free from runtime conflict. Therefore, there is no need to bother the residents of smart home to resolve on-line conflicts.

• *Flexibility of control.* The adaptation engine could adapt to user goals and goal priority. Once the user goals or goal priorities change, the adaptive engine should be regenerated automatically. it should provide a certain flexibility to the control

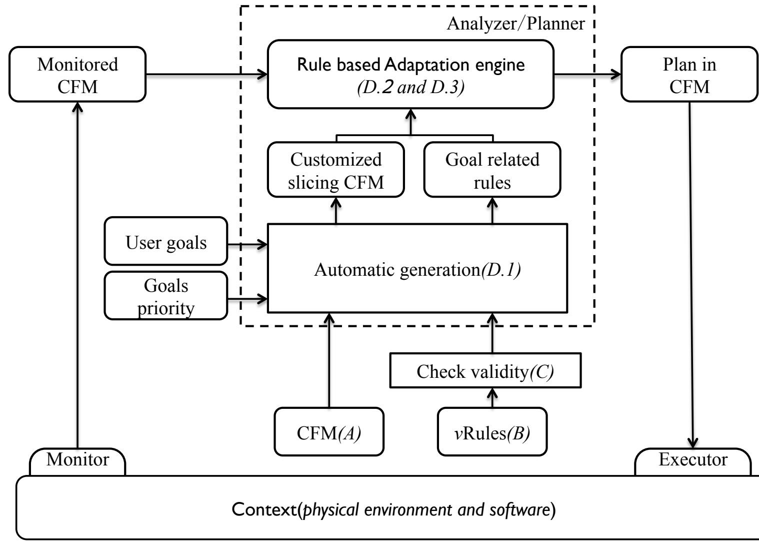

Figure 2: Overview of νRule-based Adaptation Framework

mechanism that links the capability of monitoring the context and the capability of reconfiguring the devices .

• *Integration of human preference.* Once the user modifies the states or properties of certain devices, the modification would be preserved, and not at the expense of sacrifice other selected user goals.

• *Easy for evolution.* It will be free for the user to add new features or new rules into the adaptive engine. The new added rules can be validated and added into the original rules set in a conflict-free way.

3 νRule-based Adaptation Framework

To establish aforementioned framework in Section 2, we propose a νRule-based adaptation framework, which is depicted in Figure 2. The adaptation cycle follows the traditional MAPE-K Loop [4], which stands for *Monitor*, *Analyze*, *Plan* and *Execute* based on a *Knowledge base*.

While, the improvement of our framework is the knowledge base. We distinguish two knowledge bases in this framework, i.e., the static knowledge base that is built in the development time and the dynamic knowledge base that is dynamically constructed in terms of the static knowledge base according to the current user goal setting. Owe to the characteristics of νRule that will be detailed in Section 3.2, the *Plan* part has the capability of combining the rule-based plan and the goal-based plan together. So, the run-time adaptation can response the changes in both the environment and the user goals.

The main points of our framework can be summarized as follows:

It basically follows the traditional MAPE loop of a dynamic adaptive system [4], consisting of monitor, analyzer, planner and executor.

3.0.1 Monitor and Executor

The monitor is realized by a set of sensors, which is, for example, to collect physical information such as outside temperature and room properties as well as information of devices in the smart room. According to the results of the monitor, the context (model) will be configured with specific values to form the monitored context (model), which would be further used by Analyzer and Planner to make a plan. The executor is to enforce the plan derived from Planner to implementation through the architecture level.

3.0.2 Analyzer and Planner

The adaptation engine in our framework plays the role of analyzer and planer in the MAPE loop in the sense that it filters and analyzes information given by the monitored context and work out an adaptation plan in the requirements. Our framework generates the adaptation engine automatically by a three-step process that integrates user goals and adaptation rules together.

3.0.3 Static Rule Checking

Our framework is equipped with a validity checking mechanism that guarantees the well-structured νRules free from the possibility of conflicts, and the validated rules will be further used for generation of conflict-free adaptation engines.

In the rest of this section, we will elaborate the core of our framework: (A) context feature models to capture running contexts, (B) formal definition of view-based adaptation νRules, (C) the static rule validity checking algorithm, and (D) generation of adaptation engine, as denoted in Figure 2.

3.1 Context Feature Model

For an adaptive system, its context consisting of the dynamic changing physical environment, and all the devices and applications running in the environment.

We employs feature model in our framework to depict and specify the context under consideration and establishes the context feature model (abbr. to CFM). Feature models have been adopted due to the following two reasons:

- First, feature models provide intuitive ways to express variation points and constraints, and widely adopted as variability management model in the literature of software product line ([9], [10]).
- Second, each feature in the feature model can be mapped to one or more components in the architecture [11], and thus it's easy to transform a plan

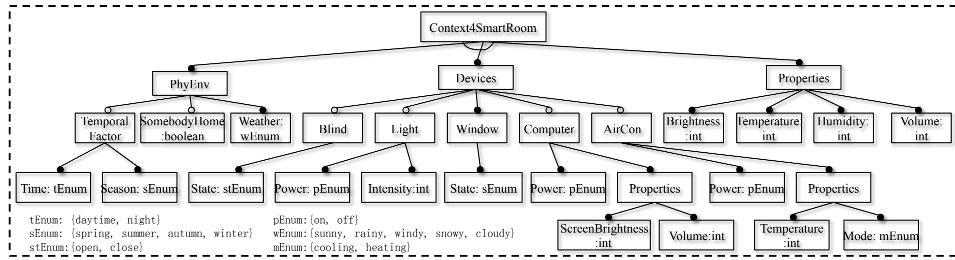

Figure 3: Context Model

expressed in feature model to a corresponding one in the architecture ([12], [13]).

Figure 3 shows a feature model for the context of a smart home system, in which, features are hierarchically organized in a tree-like structure through refinement relationship, optional feature is denoted with a small white circle on top of the feature, whereas mandatory features with a small black circle on top of the feature. A feature is mandatory if it must be selected whenever its parent is selected.

For the smart home system, its context is comprised of the physical environment and all devices equipped in this home. The physical environment consists of *temporal factor*, *weather factor*, *user presence* and *room properties*. *temporal factor* further includes *time* and *season*, whereas *room properties* includes *volume*, *brightness*, *humidity* and *temperature*. These features could be configured with various values monitored from the sensors. Considering the feature *temperature*, its value could be one collected by a thermal sensor which reflect the current temperature. Different from these features with continuous values, some features can only have enumerated states. For example, *season* only has four possible values: *spring, summer, autumn and winter*, Which can also be specified as sub features of *season*.

Another source of the contextual information is from devices. To each device, all the other devices can be regarded as its context, and its corresponding feature model is only a subtree in the whole context feature model.

This whole context feature model represents all the possible states of the smart home system. A legal configuration of the feature model, consisting of a set of selected features and their relationship, describes a valid state of the context. For example, the features set of *context, external environment, temporary factor, time, day, user presence, weather, sunny, room, light intensity, electric poser, other devices, light, power on, blind, open*, describes a valid state of the whole smart home. Once certain features are reconfigured, the context state will migrate from one to another.

3.2 νRule: View-based Adaptation Rule

One key contribution of this paper is a novel way to structure rules by refining the rule into view-based adaptation rule (νRule), which contains four parts, an

observable state view (v) of a component (device, environment), a conjunction of conditions (C), an unordered sequence of actions (A), and a set of tagged goals (G). The concrete syntax of νRule is shown in Figure 4.

view-based rule	$\nu Rule$	$::=$	$v \vdash_G C \Rightarrow A$
view	v	$::=$	fb
feature binding	fb	$::=$	$feature = value$
		$\mid$	$feature = value\ interval$
conditions	C	$::=$	$c_1 \wedge c_2 \wedge \ldots \wedge c_m$
condition	c	$::=$	fb
action sequence	A	$::=$	$a_1 ; a_2 ; \ldots ; a_n$
action	a	$::=$	$feature := value$
goals	G	$::=$	$\{g_1 , g_2 , \ldots , g_p\}$
goal impact	g	$::=$	$name : impact$

Figure 4: Syntax of νRule

The νRule rule shows that if a component is of a state v, an action A should be taken under the condition C for the purpose of G and preservation of state v. fb is called feature binding which has two alternatives: $feature$ can be either assigned with a $value$ or a $value\ interval$. For example, Light.brightness = 20 means the brightness of the light is 20, Light.brightness = (10, 30] equals to 10 $<$ Light.brightness $\leq$ 30. Each action a is a non-incremental assignment which assigns a constant $value$ to a $feature$. Each rule is bound with a list of goals. The $impact$ for each goal g specifies the actual effect for the specific state v. Let us give a νRule example:

$$\text{(R4) } Light.Power = off \vdash_{\{sl\,:\,-5, es\,:\,+3\}} Time = daytime \wedge Blind.State = close \Rightarrow Blind.State := open;\ Window.State := open$$

R4 declares that (1) if the light is power off, for the purpose of "suitable lighting" (sl) and "energy saving" (es), we should open the blind and the window if it is in daytime with the blind being closed. (2) the fact of the Blind and the Window are open implies the fact of the light is off in daytime and the blind is closed implies that someone is at home.

Generally, the νRule implies that the component state is a representative *view* of a system state (after adaptation), which will be thus called *view-based* adaptation rule.

> *The point is the use of the idea of "view" in the rule specification. Rather than showing how to propagate changes (out), the new rules specify how a component state can be achieved (in) through changes of necessary components.*

Note that in the condition of a νRule, we do not support *or* operation, this would

not weaken the expressiveness of νRule as a rule with a c_1 *or* c_2 condition equals to two νRules with conditions c_1 and c_2 separately.

$$
\begin{array}{lcl}
[\![f = val]\!] & = & \{(f, val)\} \\
[\![f \in ival]\!] & = & \{(f, ival)\} \\
[\![f := ival]\!] & = & \{(f, ival)\} \\
[\![c_1 \wedge c_2 \wedge \ldots \wedge c_m]\!] & = & [\![c_1]\!] \uplus [\![c_2]\!] \uplus \ldots \uplus [\![c_m]\!] \\
[\![a_1 ; a_2 ; \ldots ; a_n]\!] & = & [\![a_1]\!] \uplus [\![a_2]\!] \uplus \ldots \uplus [\![a_n]\!]
\end{array}
$$

Figure 5: Representation of νRule.

Even though the $\nu Rule$ is easy for developer to write view-adaptation rules, in order to clarify properties of $\nu Rule$ clearly, we interpret $\nu Rule$ using sets as shown in Figure 5. A feature binding *fb* is represented as $\{(f,\ val)\}$ which means the status of feature f is val. The operator $\uplus$ is the same as set union except that when a feature appears in both, they will be merged in the way of: (1) if the value of both are val and equal, then either is ok; (2) if the value of both are $ival$, the common interval are chosen as the result; (3) if one is a val and another is a $ival$ and the val belongs to the $ival$, then the val is the union result.

Definition 3.1 (Invariant of $\nu Rule$)**.** *A $\nu Rule$ $v \ \vdash_G \ C \Rightarrow A$ is invariant if the state of v is preserved after execution. That is, let $\{(f,\ val)\}$ = $[\![v]\!]$. For each (f', s) $\in [\![A]\!]$, if f = f', then $s \in val$.*

Note that s is always a value, while v can be either a value or value interval. For simplicity, it is considered as a special case of value interval which has only one value when v is a value.

If a rule r is invariant, the observable state v should be compatible with the action A under a condition C. Thus the rule R4 described before shows that under the goal of "suitable lighting" (sl) and "energy saving" (es), both blind and window are open implies the light is power off.

In other words, actions in A shall not change the state of v. For example, the following rule is not a proper view-based update rule.

(R5) $Light.Power = on \ \vdash_{\{sl\ :\ +5\}}$
$Time = daytime \ \wedge Blind.State = open$
$\Rightarrow Light.Power := off$

The light is on, while according to the condition that if it is daytime and the blind is open, an action of power off will be done. This violates the definition of well-behavedness of $\nu Rule$, even though the rule itself makes sense. We could rewrite this rule to a proper one by considering blind as the view:

(R5') $Blind.State = open \ \vdash_{\{sl\ :\ +5\}}$
$Time = daytime \ \wedge Light.Power = on$
$\Rightarrow Light.Power := off$

Lemma 3.2 (Rule Stability)**.** *Let r be a correct view-based adaptation rule, m be a feature model and $r(m)$ means executing rule r on feature model m resulting in a new feature model. We have $r(r(m)) = r(m)$. We call such rule r is stable.*

This means using the same rule to consecutively update feature model m twice shall be the same as that of updating only once which is obvious since (1) each action a is atomic and assigns a value to the feature. (2) executing an action a twice acts as only once as $[\![a;a]\!]$ = $[\![a]\!]$, so we have $[\![A;A]\!]$ = $[\![A]\!]$. The stability of a rule is important for self adaptive system as any times of this rule application shall not change the system. And it is also related to goals, the contribution of twice execution of the rule for a goal shall be the same as once.

Compared with the traditional rules, the view-based adaptation rules have the following characteristics:

- It is simple and expressive: view is constructed with a single feature from the whole context feature model, which reduces the design complexity of rules for developer. And a bigger and complex view can be constructed through production of simple views.
- It is invariant and stable: these two properties are important for simplifying validity checking. Each νRule is associated with one or more goals which are used for goal-based optimal rule selection.

3.3 Rule Validity Checking

In the previous section, we have discussed the invariant and stability of $\nu Rule$. In this section, we will give an algorithm for checking the well-behavedness (will be shown later) of the whole rule set.

Definition 3.3 (Order Independence)**.** *Let r_i, r_j be two different $\nu Rules$, m be a feature model. Rules r_i and r_j are said to be called order independent if the execution result of r_i followed by r_j on the model m is the same as r_j followed by r_i.*

If two rules are order independent, their effects on features are either none-overlapped or the same. For a set of rule R, we say it is confluent if the execution result of all rules in R is always the same regardless how they are applied. The stability of R means twice execution of R is the same as only executed once.

Theorem 3.4 (Well-behavedness)**.** *Let R be a finite set of rules, r_i and r_j are two rules in R, $i \neq j$. If the following two conditions are satisfied: 1. each rule r is stable; 2. rules in R are order independent. R is confluent and stable.*

This theorem is easy to prove by induction on the size of R.

Validity checking of rules contains two aspects: *a*) stability of rule set R; *b*) confluence of rule set R. We omit the algorithm for checking the stability of set

R as it is implied by stability of each rule r in R and order independence of all rule pairs.

The confluence of R is done by checking whether rules in R are order independent. Thus checking all the rule pairs: if all the pairs are not conflict (order independent), the whole set of rules are not conflict (order independent). Whether a pair of rules is order independent or not is done by the following Algorithm 1.

Algorithm 1: Order Independence Checking of Rule Pair

---

input : rule pair (r_i, r_j)
output : true, or suggestion for resolving conflicts

1. represent v_i, C_i, A_i, v_j, C_j *and* A_j *as* (f_{vi}, s_{vi}), $[\![C_i]\!]$, $[\![A_i]\!]$, (f_{vj}, s_{vj}), $[\![C_j]\!]$ *and* $[\![A_j]\!]$
2. AC_{ij}= *ft*($[\![A_i]\!]$) ∩(*ft*($[\![C_j]\!]$) ∪ *ft*($\{(f_{vj}, s_{vj})\}$))
3. AC_{ji}= *ft*($[\![A_j]\!]$) ∩(*ft*($[\![C_i]\!]$) ∪ *ft*($\{(f_{vi}, s_{vi})\}$))
4. A_{ij}= *ft*($[\![A_j]\!]$) ∩ *ft*($[\![A_j]\!]$)
5. **case** (AC_{ij}, AC_{ji}) **of**
  ($\emptyset$, $\emptyset$) → *checkAction*(r_i, r_j, A_i, A_j, A_{ij})
  (AC_{ij}, $\emptyset$) → *checkConsistency*(r_i, r_j, A_i, C_j, AC_{ij})
  ($\emptyset$, AC_{ji}) → *checkConsistency*(r_i, r_j, A_j, C_i, AC_{ji})
  (AC_{ij}, AC_{ij}) →
      checkConsistency(r_i, r_j, A_i, C_j, AC_{ij})
      checkConsistency(r_i, r_j, A_j, C_i, AC_{ji})

function *checkConsistency*(r_i, r_j, A_i, C_j, AC_{ij}){
  foreach feature element ac_{ij} of AC_{ij} **do**
    if(value($[\![A_i]\!][ac_{ij}]$) $\neq$ value(($[\![C_j]\!] \cup [\![v_j]\!]$)$[ac_{ij}]$)
    then output conflict:($[\![A_i]\!][ac_{ij}]$, ($[\![C_j]\!] \cup [\![v]\!]$)$[ac_{ij}]$)
      suggestion for r_j: ($[\![C_j]\!] \cup [\![v_j]\!]$)$[ac_{ij}]$ }
function *checkAction*(r_i, r_j, A_i, A_j, A_{ij}){
  foreach feature element a_{ij} of A_{ij} **do**
    if(value($[\![A_i]\!][a_{ij}]$) $\neq$ value($[\![A_j]\!][a_{ij}]$)
    then output conflict:($[\![A_i]\!][a_{ij}]$, $[\![A_j]\!][a_{ij}]$)
      suggestion for r_i: $[\![C_j]\!] \cup [\![v_j]\!]$
      suggestion for r_j: $[\![C_i]\!] \cup [\![v_i]\!]$ }

Algorithm 1 checks whether a rule pair is order independent or not. If not, it will give suggestions for resolving conflicts. The input of this algorithm is a rule pair (r_i, r_j). Firstly it represents conditions and actions into sets. AC_{ij} is a set of features of join of A_i and $v_j \cup C_j$. If no common features in A_i and ($v_j \cup C_j$) , this means the execution of r_i would not affect the execution of r_j, so it is the same for AC_{ji}. If both AC_{ij} and AC_{ji} is empty, we only need to check the common features in A_i and A_j have no conflict by calling function *checkAction*; if either AC_{ij} or AC_{ji} is not empty, calling function *checkConsistency* to output the conflicted feature and give suggestions for resolve conflicts.

Function $checkConsistency$ finds out all the possibilities that an action a_i in A_i could make a condition c_j in C_j become false, thus rule r_j will no longer be executed.

Let us again recall R1 and R3 shown in Introduction (Section 1) in νRule form:

$$
\begin{array}{ll}
\text{(rR1)} & \mathit{Blind.State} = \mathit{close} \vdash_{\{brightness:-3\}} \\
 & \mathit{SomebodyHome} = \mathit{true} \wedge \mathit{Time} = \mathit{daytime} \\
 & \quad \Rightarrow \mathit{Light.Power} := \mathit{on} \\
\text{(rR3)} & \mathit{Time} = \mathit{daytime} \\
 & \vdash \mathit{Weather} = \mathit{sunny} \\
 & \quad \Rightarrow \mathit{Light.Power} := \mathit{off}
\end{array}
$$

According to Algorithm 1, $[\![C_1]\!]$ = $\{$(SomebodyHome, true), (Time, daytime), (Blind.State, close)$\}$, $[\![A_1]\!]$ = $\{$(Light.Power, on)$\}$, $[\![C_3]\!]$ = $\{$(Weather, sunny), (Time, Daytime)$\}$ and $[\![A_3]\!]$ = $\{$(Light.Power, off)$\}$. AC_{13} = $\emptyset$, AC_{31} = $\emptyset$ and A_{13} = $\{$(Light.Power, off), (Light.Power, on)$\}$. Then it calls the $checkAction$ function and find out Light.Power is *on* in one rule and *off* in another rule, thus rR1 and rR3 are conflict. We could give one possible revision of rules by restructuring rR1 and rR3 as follows:

$$
\begin{array}{ll}
\text{(rrR1)} & \mathit{Blind.State} = \mathit{close} \vdash_{\{brightness:-3\}} \\
 & \mathit{SomebodyHome} = \mathit{true} \wedge \mathit{Time} = \mathit{daytime} \\
 & \quad \Rightarrow \mathit{Light.Power} := \mathit{on} \\
\text{(rrR3)} & \mathit{Time} = \mathit{daytime} \\
 & \vdash \mathit{Weather} = \mathit{sunny} \\
 & \quad \Rightarrow \mathit{Light.Power} := \mathit{off}
\end{array}
$$

3.4 Generation of Adaption Engine

The adaptation engine, which is used as our analyzer and planner in the adaptive system, is automatically generated by a three-step process: goal-based configuration, $\nu Rule$ implementation and reconfiguration of CFM.

3.4.1 Goal based Configuration

This step obtains a slice of context feature model and then configures it according to user goals by an algorithm. The feature model slice (*CFMS*) comprises goal-related features and supplementary information, aiming to provide a sub feature model relate to user goals. This *CFMS* is then configured for the optimization of overall goal satisfaction. In the process of slice and configuration, step 1 generates a collection of goal-related νRules for further rule-based adaptation. Here, a νRule is said to be goal-related if it's tagged by one or more customized goals, and a feature is said to be goal-related if it acts as the view feature in more than one goal-related νRules.

Step 1 takes inputs from two sources: one is from developers at the offline phase, and the other is from residents' daily life dynamically. The first input comprises a context feature model (*CFM*) and a collection of validated νRules, which we have already introduced in Section 3.1 and 3.2 respectively. The second input includes the set of customized user goals and the priority of these goals. The set of customized user goals are obtained from the user goal model, through a process of goal customization. In this example, we assume the residents has concern about goals of {*energy saving (es)*, *room brightness (rb)*, *computer screen brightness (csb)*, *thermal comfort (tc)*, *home security (hs)*} and no interest in goals like *acoustic comfort* and *indoor air quality* . All the concerned goals constitute a set of customized user goals. The other dynamic input is goal priority, which reflects the relative importance of the customized user goals. The priority of user goals are expressed here using a vector of weights , each of whose values denotes how much influence the corresponding goal has on overalll user satisfaction, i.e.

$$\{\omega_{es}, \omega_{rb}, \omega_{csb}, \omega_{tc}, \omega_{hs}\} = \{0.2, 0.3, 0.1, 0.1, 0.3\}$$

User goals and goal priorities, as two inputs of this step, might change dynamically. Once either user goals or goal priorities have changed, step 1 will be performed iteratively.

Algorithm 2 realizes the function of step 1 by first picking out a collection of goal related νRules. For the above example, 4 goal related νRules in this set are as follows, and they are tagged by *rb*, *csb* or *es* respectively.

(GR1) $\mathit{Light.Power} = \mathit{on}$
$\quad \vdash_{\{rb:+3, es:-3\}} \mathit{Time} = \mathit{daytime} \wedge \mathit{Weather} = \mathit{sunny}$
$\quad\quad \Rightarrow \mathit{Blind.State} := \mathit{close}$

(GR2) $\mathit{Blind.State} = \mathit{close}$
$\quad \vdash_{\{rb:-2\}} \mathit{Time} = \mathit{daytime} \wedge \mathit{SomebodyHome} = \mathit{true}$
$\quad\quad \Rightarrow \mathit{Light.Power} := \mathit{on}$

(GR3) $\mathit{Computer.Properties.ScreenBrightness} = 1$
$\quad \vdash_{\{csb:+1\}} \mathit{Time} = \mathit{daytime} \wedge \mathit{Weather} = \mathit{sunny}$
$\quad\quad \Rightarrow \mathit{Blind.State} := \mathit{open}$

(GR4) $\mathit{Computer.Properties.ScreenBrightness} = 1$
$\quad \vdash_{\{csb:+1\}} \mathit{Time} = \mathit{daytime} \wedge \mathit{Weather} = \mathit{cloudy}$
$\quad\quad \Rightarrow \mathit{Blind.State} := \mathit{open} \wedge \mathit{Light.State} = \mathit{on}$

Algorithm 2 then constructs a feature model slice (*CFMS*), a sub model of the context feature model (*CFM*). Goal related features, extended features, together with their interrelationship constitute *CFMS*. First, goal related features will be added into the *CFMS*. Since g are already tagged on each νRule, the algorithm can find the goal-related features by simply examining the collection of goal related νRules and extracting their υ part. Secondly, the extended features and refinement relationship will be added to the *CFMS* as supplementary information, where extended features denote those features that have refinement relationship with the goal related features,

Algorithm 2: Goal Oriented Selection and Configuration

```
input  : νRules νRules,context feature model CFM,
         customized user goals goals, goal priority priority
output : configured CFM slice configured-CFMS, goal-related νRules
         gνRules,

gνRules = ∅;
features = ∅;
FtoGImpacts = ∅;
for each νRule in νRules do
    if νRules.getGoals() ∩ goals ≠ ∅ then
        gνRules = gνRules ∪ νrule;
        νFeature = rule.getνFeature();
        features = features ∪ νFeature;
        for each g in νRule.getGoals do
            impact = νRule.getImapctFor(g);
            fgimpact = (νFeature, g, impact);
            FtoGImpacts = FtoGImpacts ∪ fgimpact;
        end
    end
end
extendedFeatures = ∅;
refinementRelation = ∅;
for each feature in features do
    fArray = CFM.getExtendedFeatures(feature);
    extendedFeatures = extendedFeatures ∪ fArray;
    refinementRelation = refinementRelation
    ∪CFM.getRelation(feature, fArray);
end
SFM = features ∪ extendedFeatures ∪ refinementRelation;
s_ogs=calOverallGoal(features,goals, priority, FtoGImpacts);
while time<threshold do
    for each feature in features do
        currentState=feature.getState();
        feature.changeState();
        S'_ohs=calOverallGoal(features,goals, priority, FtoGImpacts);
        if S_ohs≥S'_ohs then feature.setState(currentState);
        else S_ohs=S'_ohs;
    end
end
```

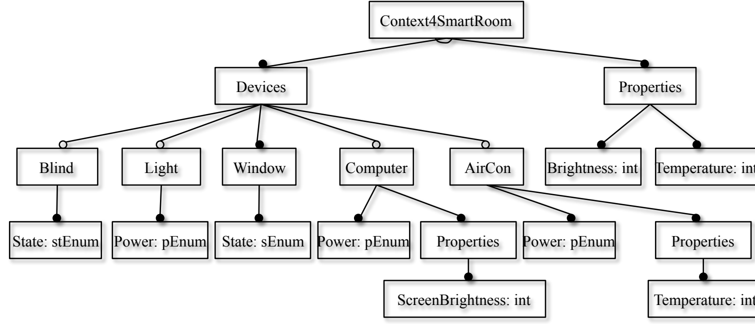

Figure 6: Feature Model Slice

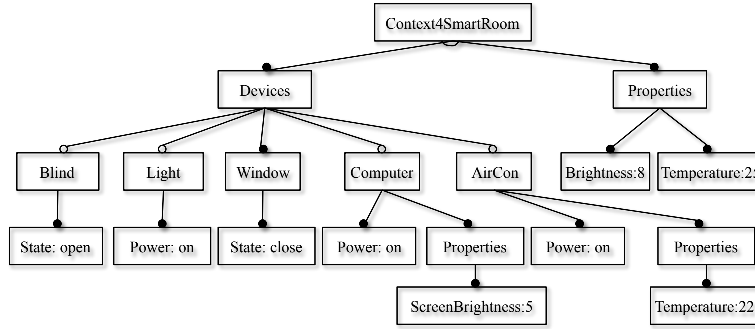

Figure 7: Configured Context Feature Model Slice

e.g. the parent feature *Computer* of *computer.screenBrightness*. Figure 6 is one *CFM* slice for the running example. With this model slice, users can concentrate on those goal related features rather than the whole feature model, and easily determine whether the context state coincides with his or her wishes.

After having obtained the *CFMS*, Algorithm 2 configures it to optimize the overall goal satisfaction S_{ogs}.

$$s_{ogs} = \sum_{g \in G} \left(Imp_g \times \omega_g \right) = \sum_{g \in G} \left(\left(\sum_{f \in F} Imp_{ftog} \right) \times \omega_g \right)$$

In this function, Imp_g denotes the impact for a specific goal, which is the sum of impact from current features' status. The configuration uses a mountain climbing algorithm to optimize s_{ogs} by calling $calOverallGoal$. It keeps changing the $state$ of features within a given time threshold, and calls $calOverallGoal$ to calculate a new s_{ogs}. The change of $state$ will only be accepted if it leads to a larger s_{ogs}, otherwise the change will be discarded. The configured $CFMS$ as figure 7 shows.

3.4.2 Implementation of View-based Adaption Rules

One important part of the whole view-update framework is the adaptation engine for executing $\nu Rule$, which is implemented by employing the powerful bidirectional transformation languge BiFlux [14]. BiFlux proposed a new programming by update paradigm which is different from all existing approaches that it let programmer have flexibility to write bidirectional transformations as intentional updates, while getting bidirectional transformations for free. Since the rules are described as an update of the context feature model, the translation from νRule to update is quite straight-forward.

The implementation of νRule by BiFluX including two parts: representation of a view feature model which only specifies one feature and a context feature model which includes all features monitored from Context, not only every device in the environment such as computer, TV and air conditioner etc, but also the real environment like temperature, weather and moisture; and translation of $\nu Rule$s as BiFluX updates.

The context is represented using the widely-used format XML and the document type definition language DTD for defining the data structure. All well-structured adaptive rules are translated into updates in BiFluX. As $\nu Rule$s have been explained, let us describe the translation from rule to BiFluX update by using the rule R4 in section 3.2.

The goals suitable lighting (*sl*) and energy saving (*es*) specified in this rule have been used in section 3.4.1 for choosing a set of rules to satisfy user's goals, thus we remove it when translating rule into update. For simplicity, we omit the definition of both source and target DTD, the XML representation of context feature model and view feature model. The translated update is shown in the following.

```
PROCEDURE updateFM($fm:FeatureM, $v:FeatureV) =
UPDATE $fm BY {
 MATCH -> REPLACE IN $fm/Devices/Blind/State WITH 'open;'
          REPLACE IN $fm/Devices/Window/State WITH 'open;'
}
FOR VIEW $lightPower IN $v/Light
WHERE $fm/PhyEnv/TemporalFactor/Time = 'daytime'
 and $fm/Devices/Blind/State = 'close'
 and $lightPower = off
```

This program means updating feature model \$fm which corresponds to a DTD named *FeatureM* by one feature \$v: if the view feature \$v satisfies the condition that the light power is off , then check whether it is daytime and the Blind is closed. If both are true, update the status of Window and Blind in feature model \$fm to 'open'. $\nu Rule$s translated to BiFluX are automatically checked against well-behavedness [14] of bidirectional transformation which further guarantees the correctness of the $\nu Rule$s.

3.4.3 Reconfiguration of *CFM*

Step 3 reconfigures the current context feature model. The *CFM* has been configured through monitor and filter procedure, demonstrating the information of running context in feature level. Then it is reconfigured through step 3, and the reconfigured model acts as an adaptation plan in feature level. The reconfiguration process constitutes two parts: reconfiguration based on *CFMS* and reconfiguration based on νRules.

The first time of reconfiguration is based on configured *CFMS* from step 1. This configured *CFMS* comprises a collection of (*feature*, *state*) pairs to be maintained. For synchronization, features in context feature model will first be configured according to their states in the feature model slice. Through first time of configuration, goal-related features in the context feature model are configured.

The second time of reconfiguration is based on νRules. Step 1 has generated a collection of goal-related νRules for further rule-based adaptation. Since a νRule specifies how to maintain a state, this step will activate a subset of νRules to maintain (*feature, state*) pairs for goal-related features. The pair (*Computer.screen-Brightness,1*) needs to be kept according to configured smart-room *CFMS*, so the νRules GR3 and GR4 (shown in 3.4.1) are activated for runtime adaptation.

An activated rule will only be performed when its condition is evaluated to be true. The condition of rules are evaluated on the current configuration of *CFM*. When the condition of GR3 ($Time = daytime \wedge Weather = sunny$) is true, the action part of this rule will be performed to set the state of blind *open*. When physical environment migrates, changes will be monitored and condition of rules will be reevaluated. Condition of GR4 ($Time = daytime \wedge Weather = cloudy$) will be evaluated to be true when weather turns into cloudy. Afterwards action part of *GR4* will be performed to configure feature states.

4 User Preferences

On the basis of MAPE loop, we have made two important extensions to support dynamic user requirements. One extension is about the integration of user preferences, and the other is about the evolution of νRules (shown in Figure 8). To integrate user preferences into MAPE loop, we provide an interface for the users to read and update. On the other hand, the users could add new rules to the set of νRules, and validity check will be performed to eliminate conflicts.

4.1 Integration of User Preference

The comprehensive system acts autonomously towards the system goals and assists the users to reach their preferred states. Since different users do not expect a same adaptation results, we provide an interface for the users to modify according to their personal preferences. This modification will be kept to satisfy users, resulting in

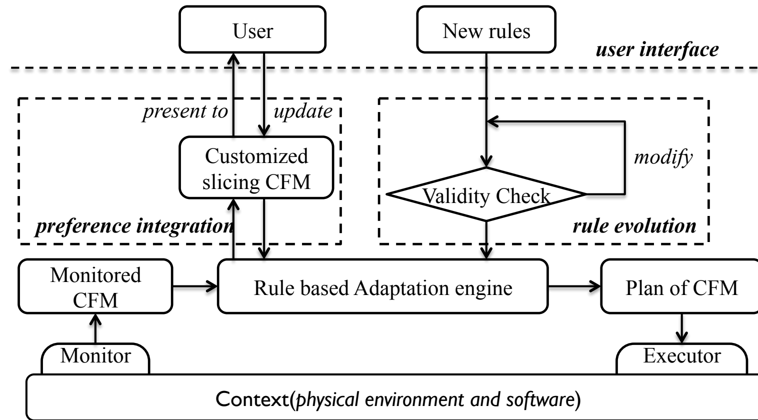

Figure 8: Integrate MAPE with Dynamic Requirements

the reconfiguration of some other features. Modification could be performed in two levels: feature level and architecture level.

4.1.1 Modification in Feature Level

The *feature model slice* in the *adaptation engine*, which consists of goal-related features and supplementary information, will be provided to users through an interface. The users can justify whether they are satisfied with the current configuration of *CFMS*, and make feature level modification for personal likings. They could change the *state* of *feature* f from S_1 to S_2 if they prefer (f, S_2).

Through the modification, user preferences have been integrated into the reconfigured *CFMS*. This reconfigured *CFMS* will be used for further rule-based adaptation. Our νRules have defined what$(C : A)$ is necessary to keep a state for the view feature. Therefore, νRules which embed (f, S_2) will be activated to keep the modified status for feature f.

4.1.2 Modification in Architecture Level

Our framework supports user intervention. The users could conduct modification in architecture level directly, i.e. the position of the blind, the power state of the appliances and the volume of multimedia devices.

These modification will be traced to modify the state of corresponding features. If the corresponding feature is not a goal-related one, it won't cause further modification. On the other hand, if this feature is a goal-related feature, it will lead to further reconfiguration and adaptation. First the current $CSFM$ will be reconfigured through trace and analysis.Then the related nuRules will be activated to maintain this reconfiguration, guaranteeing that the modification will not be revoked and it is not at the expense of the collision of other goals.

4.2 Rule Evolution

At the run time phase, the νRules set does not stay static and it has a possibility of evolution. Since the incomplete knowledges at design phase, new features might be discovered and new rules might be added into the νRules set. Our framework has a good scalability and could well support rules evolution.

For example, there is a νRule in the set.

$$(R6)\ airCon.Power = \mathit{off}\ \vdash_{\{es:+3,tc:-3\}}\ Temperature > 27 \\ \Rightarrow\ Window.State := open$$

Afterwards, a new feature *rainy'* has been discovered on the run-time phase, and the users add a νRule related to rainy into the νRules set.

$$\text{(R7)}\ airCon.Power = \mathit{off}\ \vdash_{\{es:+3,tc:-3\}}\ Weather = rainy \\ \Rightarrow\ Window.State := close$$

When the goal es or tc is customized, conflict will arise between the νRules $R6$ and $R7$. This conflict is caused by the implement information in the *condition* part of $R6$. Therefore, we modify $R6$ to add additional condition $\overline{rainy}$.

$$\text{(R6')}\ airCon.Power = \mathit{off} \\ \vdash_{\{es:+3,tc:-3\}}\ Temperature > 27 \wedge \overline{Weather = rainy} \\ \Rightarrow\ Window.State := open$$

This modification eliminates the conflicts between the two rules. The new νRule $R7$ are added into the set, and the old one $R6$ are updated and become more reasonable.

5 Threats to Validity

This section discusses some important factors that must be considered in our adaptation framework, and potential threats that might affect the validity of our work, as well as how they are mitigated or accommodated.

5.1 Expressiveness

Threats to expressiveness involves questions of whether our νRule-based adaptation framework is powerful enough to express any user goal driven adaptation.

The core of the framework is the νRule, which has a well formed structure, with two interesting properties (i.e., invariant view state for any single rule, and order-independent between rules) that guarantee against conflicts between rules. It seems that the properties weaken the expressive power of our framework at first glance, but actually it is not the case. This is because (1) a complex adaptation rule can be constructed through combining small and simple rules, due to the structured characteristic of our νRule, as well as (2) new rules can be added with the rule

evolution mechanism provided by our framework. The capability of combination and evolution of rules makes our framework be endowed with powerful expressive ability.

5.2 Construct

Threats to construct validity involve questions of whether our adaptation framework has been implemented by using adequate methodology and technology. First of all, our framework is an enhancement to the traditional MAPE-K loop approach with the view-update rules as the adaptation knowledge, and the rules can further evolve over time in response to changes in the operating context and user requirements. Moreover, our framework employs feature model to modeling and managing the commonality and variability of the domain under consideration. Feature model, on one hand, can be easily reconfigured according to different requirements and also slices according to different goals, and on the other hand, there exist relatively mature platforms ([9], [15], [16]) to support the product generation from a feature model configuration.

5.3 Applicability to Practice

We have demonstrated our adaptation framework throughout the paper using the nontrivial case of smart home system. And we are confident that our framework can be effective in any systems that can be supported by either MAPE-loop based or rule-based approaches, though we have not yet tested it with other larger cases in practice. To mitigate the threats to its applicability, we will further study in the field, and investigate more systems by equipping them with adaptation capability using our framework.

6 Related Work

Our idea of view-based adaptation rules was greatly inspired by bidirectional transformation [17, 18, 19], a new mechanism for synchronizing and maintaining the consistency of information between input and output. Bidirectional transformations, originated from the *view updating* mechanism in the database community [20, 21, 22], have been recently attracting a lot of attention from researchers in the communities of programming languages and software engineering since the pioneering work of Foster et al. on a combinatorial language for bidirectional tree transformations [17]. Bidirectional transformations have seen many interesting applications, including the synchronization of replicated data in different formats [17], presentation-oriented structured document development [23], interactive user interface design [24] or coupled software transformation [25]. Different from the existing approach whose focus is on bidirectionalization of unidirectional transformation, we have made the first attempt to design simple but powerful view-based adaptation rules (a single-view bidirectional transformation) for systematically

constructing adaptive systems. This is a continuation of our effort in designing a powerful language to specify intentional update propagation [14].

Dynamic product line and feature models have provided new opportunity to dynamic adaptive systems. There are already some work in this area.[26] analyses commonality and differences between software product line and runtime adaptation, and gives a conclusion that it is feasible to integrate variability management in both areas. [27] introduces how to utilize parts of SPL infrastructure to adapt at runtime, and introduce that major uses of SPL in design of self-adaptive system are variability models and runtime reconfigurations.[12] [5] [13] are three important works using DSPL ideas to enable self-adaptation. They use feature models as the variability model and enable runtime reconfiguration. These work fill the gap between features and architecture by various approaches, including direct link[13], transformation rule[28], aspect model weaving and common variability language.

A lot of work uses action based rules for runtime analysis and plan ([29], [30] ,[31]), where an adaptation management system is responsible for monitoring events, evaluating conditions and initiating actions. In a rule-based dynamic adaptation system, ability to evolution is necessary to handle unanticipated situations and enhance efficiency. There are mainly two types of evolution including modifying rules and discovering new system variants. [32] which enable evolution. Our work could also well support evolution.

Our approaches are based on dynamic software product line technique ([33, 34]) and bidirectional model transformation ([35, 18, 14]) to address these issues. DPLC has provided new opportunities for self-adaptation cause it has properties of context-awareness, resource-aware decision-making, permanent service delivery, and consistent dynamic reconfiguration. There are a few researches applying DPSL techniques to support self-adaptive systems. Our approach is also based on dynamic software product line techniques and we have introduced bidirectional transformation between variability models.

7 Conclusion

In this paper, we propose a systematic view-based adaptation framework for realizing more programmable and safer self-adaptive system. In our framework, rules are well structured according to each component by defining how a component state can be achieved through changes of necessary components. With the help of view-based approach, it becomes much easier to detect collision and keep rules consistent, and the detection can be conducted at design-phase statically. In order to ensure that the adaptation rules can best realize user goals, we tagged goals and impact on νRules and we use goal models as the input for generation of adaptive engine. We make an extension to support dynamic user requirements at run-time.

The νRule based framework can be used to support dynamic adaptation. In this paper, we apply our framework only on smart room systems. We will apply it on some bigger scenario and combine it with practical software product line in the

future.

References

[1] M. Acher, P. Collet, F. Fleurey, P. Lahire, S. Moisan, J.-P. Rigault *et al.*, "Modeling context and dynamic adaptations with feature models," in *Proceedings of the 4th International Workshop Models@ run. time*, 2009.

[2] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "A language for specifying security and management policies for distributed systems," *London: Department of Computing, Imperial College, Tech. Rep*, 2000.

[3] I. Lanese, A. Bucchiarone, and F. Montesi, "A framework for rule-based dynamic adaptation," in *Proceedings of the 5th International Conference on Trustworthly Global Computing*, ser. TGC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 284–300.

[4] B. H. e. a. Cheng, "Software engineering for self-adaptive systems," B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26.

[5] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven, "Using architecture models for runtime adaptability," *Software, IEEE*, vol. 23, no. 2, pp. 62–70, 2006.

[6] J. O. Kephart and R. Das, "Achieving self-management via utility functions," *Internet Computing, IEEE*, vol. 11, no. 1, pp. 40–48, 2007.

[7] A. Van Lamsweerde, "Goal-oriented requirements engineering: A guided tour," in *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*. IEEE, 2001, pp. 249–262.

[8] C. Reinisch, M. J. Kofler, F. Iglesias, and W. Kastner, "Thinkhome energy efficiency in future smart homes," *EURASIP Journal on Embedded Systems*, vol. 2011, p. 1, 2011.

[9] K. C. Kang, J. Lee, and P. Donohoe, "Feature-oriented product line engineering," *IEEE software*, vol. 19, no. 4, pp. 58–65, 2002.

[10] D. Batory, "Feature models, grammars, and propositional formulas." in *Proc. of SPLC,*, ser. LNCS 3714. Springer–Verlag, 2005, pp. 7–20.

[11] M. Voelter and I. Groher, "Product line implementation using aspect-oriented and model-driven software development," in *Software Product Line Conference, 2007. SPLC 2007. 11th International*. IEEE, 2007, pp. 233–242.

[12] B. Morin, O. Barais, J. Jezequel, F. Fleurey, and A. Solberg, “Models@ run. time to support dynamic adaptation,” *Computer*, vol. 42, no. 10, pp. 44–51, 2009.

[13] C. Cetina, P. Giner, J. Fons, and V. Pelechano, “Autonomic computing through reuse of variability models at runtime: The case of smart homes,” *Computer*, vol. 42, no. 10, pp. 37–43, 2009.

[14] T. Zan, H. Pacheco, and Z. Hu, “Writing bidirectional model transformations as intentional updates,” in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 488–491. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591102

[15] J.-P. Tolvanen and S. Kelly, “Defining domain-specific modeling languages to automate product derivation: Collected experiences,” in *Software Product Lines*. Springer, 2005, pp. 198–209.

[16] S. Deelstra, M. Sinnema, and J. Bosch, “Product derivation in software product families: a case study,” *Journal of Systems and Software*, vol. 74, no. 2, pp. 173–194, 2005.

[17] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, “Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem,” *ACM Transactions on Programming Languages and Systems*, vol. 29, no. 3, p. 17, 2007.

[18] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, “Bidirectional transformations: A cross-discipline perspective,” in *ICMT*, ser. Lecture Notes in Computer Science, R. F. Paige, Ed., vol. 5563. Springer, 2009, pp. 260–283.

[19] Z. Hu, A. Schürr, P. Stevens, and J. F. Terwilliger, “Dagstuhl Seminar on Bidirectional Transformations (BX),” *SIGMOD Record*, vol. 40, no. 1, pp. 35–39, 2011.

[20] F. Bancilhon and N. Spyratos, “Update semantics of relational views,” *ACM Transactions on Database Systems*, vol. 6, no. 4, pp. 557–575, 1981.

[21] U. Dayal and P. Bernstein, “On the correct translation of update operations on relational views,” *ACM Transactions on Database Systems*, vol. 7, pp. 381–416, 1982.

[22] G. Gottlob, P. Paolini, and R. Zicari, “Properties and update semantics of consistent views,” *ACM Transactions on Database Systems*, vol. 13, no. 4, pp. 486–524, 1988.

[23] Z. Hu, S.-C. Mu, and M. Takeichi, “A programmable editor for developing structured documents based on bidirectional transformations,” *Higher-Order and Symbolic Computation*, vol. 21, no. 1-2, pp. 89–118, 2008.

[24] L. Meertens, “Designing constraint maintainers for user interaction,” 1998, manuscript available at `http://www.kestrel.edu/home/people/meertens`.

[25] R. Lämmel, “Coupled Software Transformations (Extended Abstract),” in *SETS 2004*, 2004.

[26] V. Alves, D. Schneider, M. Becker, N. Bencomo, and P. Grace, “Comparitive study of variability management in software product lines and runtime adaptable systems,” 2009.

[27] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, “Dynamic software product lines,” *Computer*, vol. 41, no. 4, pp. 93–95, 2008.

[28] M. Acher, P. Collet, P. Lahire, S. Moisan, and J.-P. Rigault, “Modeling variability from requirements to runtime,” in *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*. IEEE, 2011, pp. 77–86.

[29] P.-C. David and T. Ledoux, “An infrastructure for adaptable middleware,” in *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*. Springer, 2002, pp. 773–790.

[30] L. Capra, W. Emmerich, and C. Mascolo, “Carisma: Context-aware reflective middleware system for mobile applications,” *Software Engineering, IEEE Transactions on*, vol. 29, no. 10, pp. 929–945, 2003.

[31] L. Capra, “Reflective mobile middleware for context-aware applications,” Ph.D. dissertation, University of London, 2003.

[32] A. Helleboogh, D. Weyns, K. Schmid, T. Holvoet, K. Schelfthout, and W. Van Betsbrugge, “Adding variants on-the-fly: Modeling meta-variability in dynamic software product lines,” in *Proceedings of the Third International Workshop on Dynamic Software Product Lines (DSPL@ SPLC 2009)*, 2009, pp. 18–27.

[33] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch, “Using product line techniques to build adaptive systems,” in *Proceedings of the 10th International on Software Product Line Conference*, ser. SPLC ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 141–150. [Online]. Available: http://dl.acm.org/citation.cfm?id=1158337.1158688

[34] C. Ghezzi and A. M. Sharifloo, “Dealing with non-functional requirements for adaptive systems via dynamic software product-lines,” in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, Eds., vol. 7475. Springer, 2010, pp. 191–213.

[35] P. Stevens, “Bidirectional model transformations in QVT: Semantic issues and open questions,” in *Proc. 10th MoDELS*, ser. Lecture Notes in Computer Science, G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, Eds., vol. 4735. Springer, 2007, pp. 1–15.