# Modelling Adaptation Policies as Domain-Specific Constraints

Hui Song[1], Xiaodong Zhang[2], Nicolas Ferry[1], Franck Chauvel[1], Arnor Solberg[1], and Gang Huang[2]

[1] SINTEF, Norway. {first.last}@sintef.no
[2] Peking University, China. {xdzh, hg}@pku.edu.cn

**Abstract.** In order to develop appropriate adaptation policies for self-adaptive systems, developers usually have to accomplish two main tasks: (i) identify the application-level constraints that regulate the desired system states for the various contexts, and (ii) figure out how to transform the system to satisfy these constraints. The second task is challenging because typically there is complex interaction among constraints, and a significant gap between application domain expertice and state transition expertice. In this paper, we present a model-driven approach that relieves developers from this second task, allowing them to directly write domain-specific constraints as adaptation policies. We provide a language to model both the domain concepts and the application-level constraints. Our runtime engine transforms the model into a Satisfiability Modulo Theory problem, optimises it by pre-processing on the current system state at runtime, and computes required modifications according to the specified constraints using constraints solving. We evaluate the approach addressing a virtual machine placement problem in cloud computing.

## 1 Introduction

As software systems and their interactions with the executing environments are becoming more and more complex, many systems are required to adjust themselves at runtime to harmonize with their dynamic environments. Such self-adaptations can be seen as guided transitions between system states (such as the system's structure, configuration, environments, etc.). A key challenge to build such self-adaptive systems [1] is to develop the adaptation policies that guide such transitions [2,3]. To develop appropriate policies, developers usually need to cope with the following two concerns: (i) to identify the constraints on the system states for the various contexts, which determine when the system needs to be adapted, and what are the desired states after adaptation. (ii) to figure out the appropriate transitions between system states that satisfy these constraints. Developers specify these transitions in a policy language (e.g., in event, condition, action (ECA) rules). Figuring out the transitions is particularly challenging, because the constraints usually have complex *interactions* with each other, i.e., a transition that satisfies one constraint may violates another. Moreover, there is typically a conceptual gap between reasoning about

constraints in the application domain (e.g., cloud computing, health care, etc.), and the specification of transitions in a state-transition model (e.g., ECA or state machines).

In this paper, we propose a model-driven approach where developers can directly specify the constraints as adaptation policies, using concepts specific to the application domain and applying the object oriented constraint specification language, the OCL [4]. Our runtime engine then dynamically computes the required modifications on the current system to satisfy the constraints.

This approach is based on our previous work [5], which showed that SMT (Satisfiability Modulo Theory) constraint solving [6] can be used to compute adaptation decisions from constraints in First Order Logic (FOL). However, we used a simple SMT theory based on variables and operations, and therefore it only supports adaptation of numeric configurations. To support domain-specific modelling of constraints, more expressive theories to encode structural information such as objects and references between them, are needed. This implicates the challenge to transform expressive OCL constraints to SMT. Moreover, the previous work lacks a way to assist developers in specifying the constraints. To address these challenges, we present the following contributions in this paper.

- A modelling language for specifying adaptation policies based on MOF and OCL, which allows developers to first specify the relevant application domain concepts, and then the adaptation constraints applying these concepts;
- A new method to encode object-oriented models and constraints to SMT instances based on uninterpreted functions and first order logic, which enables the use of SMT constraint solving for adaptation planning.
- A new partial evaluation semantics on OCL, which realises the systematic transformation from OCL constraints to formulas that can be precessed by SMT, and optimises the formulas by embedding the current system states into them.

We have applied the approach on a representative self-adaptation problem, the dynamic mapping of virtual machines to physical machines in clouds. The case study shows that the approach is able to specify classical adaptation policies, and produces desired adaptation decisions. The partial evaluation significantly improves the performance of constraint solving, making it applicable at runtime.

The rest of this paper is organised as follows. Section 2 introduces a running example, and provides an overview of the approach. Sections 3 presents our modelling language, and Section 4 shows how to transform the models into SMT instances. In Section 5 we briefly discuss how to calculate the adaptation decision using constraint solving. Section 6 shows our case study. In Section 7 related approaches are discussed and Section 8 concludes the paper.

## 2 Approach Overview

### 2.1 Motivating example

Managment of the mapping of virtual machines (VM) to physical machines (PM) in private clouds can be treated as a dynamic adaptation problem. Different from

renting VMs from public clouds, an organisation that sets up its own private cloud has the full control of the infrastructures (i.e., PMs) behind the VMs. As the system keeps evolving (e.g., new VMs are provisioned, applications are deployed on VMs), the infrastructure administrators need to adjust the placement and configuration of VMs, in order to optimise the overall deployment.

Figure 1 illustrates a simplified private cloud, where three VMs are placed on two PMs, each VM requires and provides different numbers of CPU cores and memory sizes (unit in GB). We assume that the adaptation engine is capable of altering the VM placement and the provided CPU cores. Early approaches on VM placement mainly consider resource limitations and consolidations (e.g., a single VM's CPU core number should not exceed its hosting PM, total VM memory should not exceed the PM's capacity, and using as few PMs as possible to save energy) [7]. These concerns implies that we should migrate `vm3` to `pm1`. However, there are other concerns that should also impact the adaptation decisions. For example, based on the applications shown in Figure 1 we can see that `vm1` and `vm2` are replicated for backup purposes[3]. The two VMs should be placed in different PMs so that a physical crash would not alter both VMs making the application data unavailable. Moreover, if `vm2` and `vm3` are communicating frequently, they should ideally be deployed on two PMs that are "close" in terms of latency, or even the same PM. Considering these two objectives, a potential modification is to swap `vm1` and `vm3`, and decrease the CPU cores of `vm1`, even though this violates the objective of consolidation. Finally, migrating VMs is an expensive operation, and the cost is proportional to their memory size. When a to-be-migrated VM is big in terms of memory, a better trade-off may be to keep the current configuration.

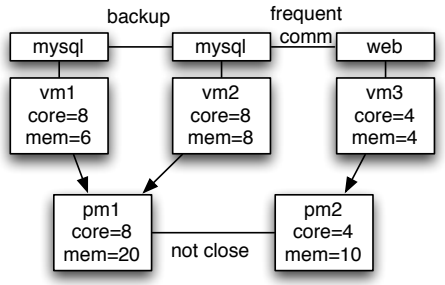


**Fig. 1.** A simplified VM placement problem

As highlighted by the example, an adaptive system typically have many constraints, and an action that satisfies one constraint may violate others. Manually developing adaptation policies by exhausting all the constraints to figure out appropriate actions is not practical for complex systems.

### 2.2 The approach

In this paper, we propose a model-driven approach to enable specifying adaptation policies as domain specific constraints. The overall architecture is illustrated in Figure 2. Developers (domain experts) apply their knowledge of the domain to specify an adaptation model. This includes the base domain *concepts*, the adaptation *capability* (i.e., what can be changed by adaptation), and the *constraints*

[3] In this figure, we simplified the identification of backup relations between VMs: Any two VMs that host applications with the same name are backup to each other

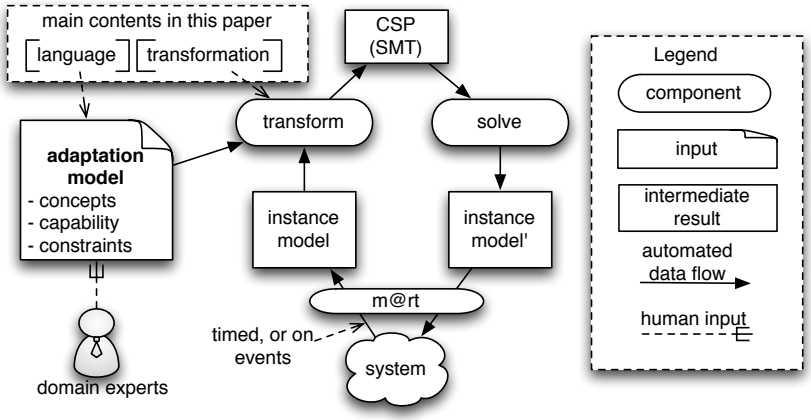


**Fig. 2.** Approach architecture

applying the domain concepts to specify what are the desired system states for various contexts. From the adaptation model, we perform an automated adaptation at runtime, using three components. The *models@runtime engine* maintains an updated instance model monitoring the system state (through a causal connection). From the current instance model and the adaptation model, the *transformation engine* interprets *what the constraints imply on the current system state*, and generates an SMT instance, which is fed to the *constraint solver* to compute the appropriate target system state, taking into account the constraints, their priorities, and the cost of system modifications. Finally, the models@runtime engine propagates the changes to the real system.

In our earlier papers, we have presented how to implement the models@runtime engine [8], and how to use constraint solving for adaptation [5]. In this paper, we focus on the *front-end* of this approach, i.e., the modelling language for domain-specific adaptation constraints, and the transformation to an SMT instance. At this stage, we base our work on the *closed world assumption*, i.e., the number of objects under each type is not subjected to be changed by the adaptation. This assumption is reasonable in our VM placement example: It is the administrator's duty to provision or terminate VMs either manually or automatically, whereas the adaptation engine takes care of optimising the deployment of them.

## 3 Constraint Modelling

We provide a prototype language based on MOF and OCL to assist the modelling of adaptation policies as constraints. Figure 3 is a snapshot of the model for the VM placement problem, taken from our text-based modelling editor with syntax checking and auto-completion. The modelling process has two steps: defining the concepts in the domain, and specifying the constraints applying the concepts.

The concept modelling is to specify the *types* of elements that compose the system states (e.g., the concepts `VMs`, `PMs` at Line 9 and 20), the *attributes* of

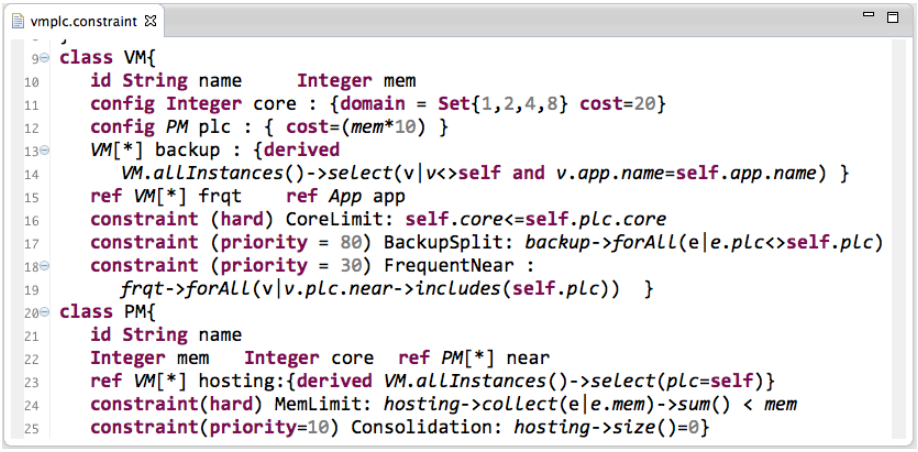


**Fig. 3.** Constraints for VM placement

these types (such as `VM.mem` at Line 10, for the memory required by a VM), and the *relations* between them (e.g., `VM.plc` at Line 12 records on which PM a VM is placed). A property (attribute or relation) marked by "[*]" is a multi-valued one. `Derived` properties can be defined via an OCL query. For example, the set of `backups` of an VM is the subset of VM instances that host `Apps` with the same name. Developers also need to define the adaptation capability of the system, including *what properties can be changed by adaptation* (which are marked by keyword `config`, such as `VM.plc` at Line 12), and *the changing scope of such a property* (which is defined by an OCL expression in type of collection, such as the domain of `VM.core` at Line 11).

Constraint modelling captures the developers' concerns regarding *what are the desired system states*, such as `MemoryLimit` (Line 24) and `BackupSplit` (Line 17), with the meanings discussed above. In our language, a constraint is defined as an boolean-typed OCL expression, inside a target class. Take the `MemLimit` constraint as an example, the direct meaning of the OCL expression is as follows: For each PM, we get the `hosted` VMs on it (which in turn is derived from the configurable reference `VM.plc`), `collect` the `mem` of these VMs, and calculate the `sum` of them. Then we claim that this sum should be less than or equal to the `mem` of the PM. Each constraint is assigned with a priority, indicating how important it is. A `hard` constraint is the one that must be satisfied. Currently, we utilise a simple priority modelling mechanism based on integer values between 0 and 100. The same mechanism is also used on the `cost` of configurable properties, which indicates how important it is to maintain the current configuration values, e.g., the cost of changing a VM's placement (i.e., VM migration) is proportional to the memory size.

## 4 SMT Instance Generation

In this section, we first give an overview about SMT, illustrated by our running example. After that, we summarise the mapping from an adaptation model to

the SMT instance. Finally, we present a partial evaluation approach to transform and simplify OCL constraints into formulas in SMT.

### 4.1 SMT overview

In order to apply automated constraint solving for adaptation, we convert the adaptation problem into an SMT instance composed of *functions* and *constraints*, based on the theories of uninterpreted functions [9], algebraic data types [10], linear arithmetic, and first order logic (FOL). An *uninterpreted function* is a function that declares domains and a codomain, but without a definition about the concrete mapping between them. A domain or codomain can be a primitive data type (integer, real or boolean), or an enumeration. Here we define enumerations using *algebraic type theory*, which supports defining a type from limited and disjoint constructors[4]. The constraints are boolean-valued FOL formulas on these functions, connected by arithmetic and logic operators. An SMT solver searches possible interpretations for the functions, satisfying all the constraints.

Figure 4 shows a sample SMT instance. It is divided in three parts.

$$\mathsf{VM} : \{vm_1, vm_2, vm_3\},\ \mathsf{PM} : \{pm_1, pm_2\},\ \mathsf{int},\ \mathsf{boolean}$$
$$vc : \mathsf{VM} \to \mathsf{int},\ pc : \mathsf{PM} \to \mathsf{int},\ vmem : \mathsf{VM} \to \mathsf{int},\ pmem : \mathsf{PM} \to \mathsf{int}$$
$$plc : \mathsf{VM} \to \mathsf{PM},\ frqt : \mathsf{VM} \times \mathsf{VM} \to \mathsf{bool},\ near : \mathsf{PM} \times \mathsf{PM} \to \mathsf{bool}$$
$$\forall vm, pm.\ (plc(vm) = pm \Rightarrow vc(vm) \leq pc(pm))$$
$$\forall pm. \left( \sum_{vm \in VM} \mathsf{ite}(plc(vm) = pm, vmem(vm), 0)) \leq pmem(pm) \right)$$
$$\forall vm, vm'. (frqt(vm, vm') \Rightarrow near(plc(vm), plc(vm'))), \forall pm. (near(pm, pm))$$
$$vc(vm_1) = 8, vmem(vm_1) = 6, plc(vm_2) = pm_1, frqt(vm_2, vm_3)...$$

**Fig. 4.** VM placement in SMT

The first part defines the realm of this VM placement problem, in the form of functions and their domain types. Under a closed-world assumption, the number of objects of type VM or PM is not subject to change by the adaptation, and therefore each type is an enumeration of its current objects (Of course, administrators can still add or remove VMs and PMs, and the new VMs can be moved between PMs by our adaptation engine). The functions for virtual cores ($vc$), physical cores ($pc$), virtual memory ($vmem$) and physical memory ($pmem$) from the enumerations to primitive types represent states of objects. The functions placement ($plc$), frequent ($frqt$) and near ($near$) represent the relations between objects. Placement ($plc$) is a *functional relation*, specifying that a VM is placed on one and only one PM. By contrast, frequent ($fqrt$) is

[4] For example, an enumeration `Colour` with `red`, `green`, and `blue` can be defined as `(type C::=r|g|b)`, meaning that the type `C` has three unique constructors, any element of `C` must belong to one of the three, and two elements are equivalent if and only if they belong to the same constructor. Algebraic type theory is supported by SMT solvers such as Z3 [11]

an example of *binary relation*. For two VMs $vm$ and $vm'$, $frqt(vm, vm') = \textbf{true}$ means they are communicating frequently.

The second part of the SMT instance defines the constraints applying FOL formulas. The first constraint specifies that the number of CPU cores required by a $vm$ cannot exceed the one provided by its hosting $pm$. The second constraint specifies that the total memory size of VMs hosted on the same PM should not exceed the memory provided by this PM. The constraint computes the sum of VM's memories utilizing a predefined ite(if-then-else) function from the SMT-LIB standard [12], which returns the second or the third parameter based on if the first one is true or false, respectively. The constraint denotes to iterate over all $vm$s, and if and only if a $vm$ is placed on the specific $pm$ its memory is added to the sum. The third constraint depicts that two frequently communicating $vm$s should be deployed to $pm$s that are *near* to each other (remark that a PM is also considered to be *near* to itself).

The third part of the SMT instance expresses the current state of the model instance as a set of equations between function calls and values. The excerpt of Figure 4 is according to the state shown in Figure 1.

Solving these SMT constraints can be very time-consuming. However, the constraints can be simplified significantly based on knowledge of the current system state (i.e., the third part), making the solving much more efficient. For example, according to Figure 1, we know that there is one and only one pair of VMs (`vm2` and `vm3`) that are frequently communicating, and this will not be changed after the adaptation. Therefore, we can weave this known information into the constraints and rewrite the third constraint in Figure 4 simply as $near(plc(vm_2), plc(vm_3))$. In our approach, we provide to directly generate such simplified constraints.

### 4.2 Mapping from adaptation model to SMT

In order to transform the adaptation model into an SMT instance, we need systematic mappings from the elements in the adaptation model to their corresponding representations in SMT. Table 1 summarises these mapping rules. A class is mapped into an enumeration, with its objects as the enumerable items, and primitive types are transformed to the corresponding integer, real or boolean types in SMT (according to `mc` in Table 1). Objects and primitive data are mapped to enumeration items or primitive values (`mo`). A `C2`-typed single-valued property `p` defined in class `C1` is mapped to an uninterpreted function with one parameter, while a multi-valued one (marked by a "*" in the table) is mapped to a function with two parameters (`mp`). The two functions $plc$ and $frqt$ in Figure 4 are examples for the two categories, respectively. The OCL constraints, and the derivation and domain definitions are mapped to FOL formulas (such as the ones shown in the second part of Figure 4). The transformation (named PE) will be shown in the next section. Finally, we generate constraints from the current values of properties (`ms`). If a single valued property $p$ of object $o$ has value $d$, we enforce $p(o) = d$. However, for a multi-valued $p$, the value will be a

set $D$ of data or enumeration items. The generated constraint enforces that for any $d_i \in D$, $p(o, d_i)$ is true, and for any other $d_j$, $p(o, d_j)$ is false.

**Table 1.** Mapping from adaptation model to SMT instance.

| model | SMT | name |
|---|---|---|
| class `C` with objects `o1`, `o2`<br>types `int, real, boolean` | enum C $\{o_1, o_2\}$<br>SMT types: $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{B}$ | `mc` |
| object `o`: `C`<br>value `v`: | enum item $o \in C$<br>literal value $v$ | `mo` |
| single property: `C1.p:C2`<br>multiple property: `C1.p:C2[*]` | function $p : C_1 \to C_2$<br>function $p : C_1 \times C_2 \to \mathbb{B}$ | `mp` |
| constraint, derivation, domain | FOL formula | PE |
| property value: `o.p=`$d$<br>`o.p=`$D$=$\{$`d`$_i \mid i \in 1..n\}$ | $p(o) = d$<br>$(\bigwedge_{d_i} a(o, d_i)) \wedge (\bigwedge_{d_j} \neg a(o, d_j)) d_j \in C_2 - D$ | `ms` |

### 4.3 Partial evaluation of OCL constraints

We use partial evaluation (PE) [13] to transform the OCL constraints into SMT predicates such as the ones shown in the second part of Figure 4, and simplify the results based on the static information in the current model instance. In particular, PE takes three inputs: an *OCL expression*, a *static model* and a *context*, and outputs FOL formulas. The following example illustrates how we notate partial evaluation in our approach.

$$[\![\texttt{self.core<self.mem-2}]\!]_{\{\texttt{self} \mapsto \texttt{vm2}\}}(m_s) = (< (\mathsf{core\ vm2})\ 6)$$

The *OCL expression* is written inside the brackets $[\![]\!]$[5]. The *static model* $m_s$ is the part of a model instance that does not change after adaptation, e.g., below is the static model corresponding to Figure 1

```
vm2:VM{mem=8, frqt=[vm3], app=[app1], core =_, plc=_}
pm2:PM{mem=10, core=4, near=[]}
```

Here we give every dynamic property (defined by `config` in Figure 3) an undefined value "_". The *context* $\tau$ is a map from variables to values or objects in $m_s$, and in this example, $\tau = \{\texttt{self} \mapsto \texttt{vm2}\}$, meaning that the variable `self` in the OCL expression represents `vm2`. The output of PE is a FOL formula, written in the standard SMT-LIB language [12] following the style of prefix notation. This output has the same meaning of $core(vm2) < 6$[6]. In this example, since

[5] We borrow the "$[\![]\!]$" notation from denotational semantics, which indicates that PE can be understood as another semantics to OCL language, i.e., a function from a static model to a FOL formula, depending on a context.

[6] We use SMT-LIB to distinguish the transformation outputs (i.e., an SMT "program") from the calculations within the evaluation. For example (`f 5`) means a SMT formula that call function `f` with parameter `5`, while `mo`($o$) means that PE obtains the enumeration item for object $o$

`vm2.mem=8` is known in $m_s$, we directly evaluate `self.mem-2` into a value 6, whereas since `vm2.core` is unknown, we translate it into a function call.

**The Fidelity Property** PE should guarantee the *fidelity* of the transformation from OCL to SMT, which means any adaptation result that satisfies the generated SMT should also satisfy the original OCL constraints, and vice versa. Specifically, a static model $m_s$ can be complemented with dynamic information (notated as $m_d$), so that $m_s+m_d$ is a complete model and therefore can be *fully* evaluated by a common OCL engine. Fidelity means that no matter what $m_d$ we give on $m_s$, $(\llbracket \mathtt{e} \rrbracket_\tau(m_s) = \llbracket \mathtt{e} \rrbracket_\tau(m_s+m_d)) \wedge \mathtt{ms}(m_d)$ is always true. Here, $\llbracket \mathtt{e} \rrbracket_\tau(m_s+m_d)$ degrades into normal OCL evaluation, and $\mathtt{ms}(m_d)$, as defined in Table 1, is a set of predicates encoding the dynamic information. For example, the sample above satisfies fidelity, because if we give `vm2.core` a value smaller than 6 (say 5), then the full evaluation results true, and $(core(vm_2) < 6 = \mathbf{true}) \wedge core(vm_2) = 5$ holds. It is the same when `vm2.core` $> 6$.

PE is executed in a recursive way, following the OCL syntax tree [4]. For example, the first step to evaluate the example above will be:
$\llbracket \mathtt{self.core<self.mem-2} \rrbracket_\tau(m_s) = (< \ \llbracket \mathtt{self.core} \rrbracket_\tau(m_s) \ \llbracket \mathtt{self.mem-2} \rrbracket_\tau(m_s))$, and after that $\llbracket \mathtt{self.mem-2} \rrbracket_\tau(m_s) = (- \ \llbracket \mathtt{self.mem} \rrbracket_\tau(m_s) \ 2) = (- \ 8 \ 2) = 6$.

In the following, we explain how we do PE by defining the transformation rules on the typical OCL syntax structures.

We start from the basic building blocks of OCL expressions. For a literal constant of primitive value, we directly transform it into the corresponding value (Equation 1). For the reference to a variable `v`, we obtain its value from the context $\tau$ and return it (2). The `let` statement (3) introduces a new variable into the main expression. We evaluate the source expression $\mathtt{e}_1$ into result $r$, and add a new variable mapping $v \mapsto r$ into the context dictionary $\tau$, so that the variable `v` in $\mathtt{e}_2$ will be $r$ in the subsequent evaluation.

$$\llbracket \mathtt{literal} \rrbracket_\tau(m_s) = \mathtt{mo}(\text{literal}) \tag{1}$$

$$\llbracket \mathtt{v} \rrbracket_\tau(m_s) = r, (v \mapsto r) \in \tau \tag{2}$$

$$\llbracket \mathtt{let\ v{=}e_1\ in\ e_2} \rrbracket_\tau(m_s) = \llbracket \mathtt{e_2} \rrbracket_{\tau \cup \{v \mapsto r\}}(m_s) \ \mathbf{where} \ \ r = \llbracket \mathtt{e_1} \rrbracket_\tau(m_s) \tag{3}$$

The transformation of OCL property calls is the main point to encode object-oriented structures into FOL formulas (4). We first evaluate the source expression `e` into result $r$. If $r$ is an object (which means that `e` purely depends on the static model), and $r.p$ has a value $v$ in the static model, we directly return $v$. However, if $r.p$ is undefined, or if $r$ is a formula (which means that `e` depends on dynamic information), we compose a new formula using the functions and constants resolved from $p$ and $r$, respectively. The composition method $pc$ (Equation 5) creates a function call if the property is single-valued. Otherwise, it enumerates all the objects $o_i$ in the property type, and creates an `ite` for each of them: if $f(s, o_i)$ is true then $o_i$ is returned, otherwise, an empty value $\perp$ is returned. We introduce an empty value $\perp$ whose semantics is that for any binary operation $*$ and value $x$, $x * \perp = x$. When the PE is finished, to obtain valid SMT formulas, we replace each $\perp$ by the corresponding default value,

such as 0 for $+$, 1 for $\times$, true for $\wedge$, etc. For example, $[\![\texttt{self.plc}]\!]_{\texttt{self}\mapsto\texttt{vm2}} = \{(\mathsf{ite}\ (=\ (\mathsf{plc}\ \mathsf{vm2})\ \mathsf{pm1})\ \mathsf{pm1}\ \bot), (\mathsf{ite}\ (=\ (\mathsf{plc}\ \mathsf{vm2})\ \mathsf{pm2})\ \mathsf{pm2}\ \bot)\}$. This result satisfies Fidelity, because if we place `vm2` to any PM, the result will be this PM plus only $\bot$, equivalent to the result of a full OCL evaluation. Equation (6) defines the derivation: the property call `e.p` will be replaced by the expression `ed` defined for `p`, with `self` redirected to the source expression `e`.

$$[\![\texttt{e.p}]\!]_\tau(m_s) = \begin{cases} v & \textbf{if } (r.p \mapsto v) \in m_s \\ pc(\texttt{mp}(p), \texttt{mo}(r), \texttt{mc}(p.type)) & \textbf{if } (r.p \mapsto \_) \in m_s \\ pc(\texttt{mp}(p), r, \texttt{mc}(p.type)) & \textbf{if } r \texttt{ is a formula} \end{cases} \quad (4)$$
$$(r = [\![\texttt{e}]\!]_\tau(m_s))$$

$$pc(f, s, c) = \begin{cases} (\mathsf{f}\ \mathsf{s}) & \texttt{single-valued} \\ \{(\mathsf{ite}\ (\mathsf{f}\ \mathsf{s}\ \mathsf{oi})\ \mathsf{oi}\ \bot) | \mathsf{oi} \in c\} & \texttt{multi-valued} \end{cases} \quad (5)$$

$$[\![\texttt{e.p}]\!]_\tau = [\![\texttt{let self=e in ed}]\!]_\tau, \textbf{if } \texttt{ed} \text{ is the derivation expression of } \texttt{p} \quad (6)$$

PE handles and simplifies structural OCL syntax rules. For an `if` expression (7), we first evaluate the condition expression $e_1$ into $r_1$. If it is determined to either **true** or **false**, we return either result of the two sub expressions. Only when $r_1$ is a formula, we transfer the OCL branch into an ite . The second sample is the binary operation (8), such as `+, ×, and` etc. If both results $r_1$ and $r_2$ of the two operands are values, we calculate the result and return it. If either $r_1$ or $r_2$ is a formula, we compose a corresponding binary operation in SMT.

$$[\![\texttt{if e}_1\ \texttt{then e}_2\ \texttt{else e}_3]\!]_\tau(m_s) = \begin{cases} r_2 & \textbf{if } r_1 = \textbf{true} \\ r_3 & \textbf{if } r_1 = \textbf{false} \\ (\mathsf{ite}\ r_1\ r_2\ r_3) & \textbf{if } r_1 \in F\_ \end{cases} \quad (7)$$
$$(r_i = [\![\texttt{e}_i]\!]_\tau(m_s), i \in \{1,2,3\})$$

$$[\![\texttt{e}_1\ \texttt{+}\ \texttt{e}_2]\!]_\tau = \lambda m_s. \begin{cases} r_1 + r_2 & \textbf{if both } r_1 \text{ and } r_2 \text{ are values} \\ (+\ r_1\ r_2) & \textbf{if } r_1 \text{ or } r_2 \text{ is a formula} \end{cases} \quad (8)$$
$$(r_i = [\![\texttt{e}_i]\!]_\tau(m_s), i \in \{1,2\})$$

We handle composite OCL syntax rules on the basis of the primitive ones above. The `collect` operation (9) is a combination of `let` (i.e., we evaluate the main expression $e_2$ repeatedly, each time with a $s_i$ from the source result). The `select` operation (10) is a combination of `if-then-else`. The resulted set will be simplified if any $r_i$ is resolved to a $true$ or $false$: For the former case, $s_i$ will be included into the resulted set, and for the latter case, it will be $\bot$. Similarly, `forAll` (11) is an extension of the binary `and` operation to multiple inputs. We divide the source set $s$ into a single element $s_h$ and the remaining set $s_t$. Then we evaluate $s_h$ and $s_t$ recursively, and combine the results by an `and`. The `sum` operation (12) is a similar extension to the binary operation `+`. We regard the `size` operation (13) as equivalent to first mapping each element to an integer 1, and then calculate the summary. The last important collection operations is `include` (14), which is often used to check if a relation holds upon two objects (e.g., Line 12 in Figure 3). We inspect the source set $r_1$ from $e_1$, and see if there

is an item related to the target value $r_2$ evaluated from $e_2$. If there is an item $r$ equal to $r_2$, the result is true; If there is an ite item in $r_1$ whose main branch equals to $r_2$, then whether $r_2 \in r_1$ depends on the condition $r_3$, and we simplify the whole operation to this condition $r_3$. Finally, we transform domain definition on a property into the following equivalent OCL expression (15): the value of `self.p` on context $\tau$ must equal to one of the values limited by `e`.

$$[\![\texttt{e}_1\texttt{->collect(v|e}_2\texttt{)}]\!](m_s) = \{[\![\texttt{e}_2]\!]_{\tau\cup\{v\mapsto s_i\}}(m_s) | s_i \in [\![\texttt{e}_1]\!]_\tau\} \tag{9}$$

$$[\![\texttt{e}_1\texttt{->select(v|e}_2\texttt{)}]\!]_\tau(m_s) = \lambda m_s.\{[\![\texttt{if } r_i \texttt{ then } s_i \texttt{ else } \bot]\!]_\tau(m_s)\} \quad s_i \in s = [\![\texttt{e}_1]\!]_\tau(m_s), r_i = [\![\texttt{e}_2]\!]_{\tau\cup\{v\mapsto s_i\}}(m_s) \tag{10}$$

$$[\![\texttt{e}_1\texttt{->forAll(v|e}_2\texttt{)}]\!]_\tau(m_s) = [\![r_h \texttt{ and } s_t\texttt{->forAll(v|e}_2\texttt{)}]\!]_\tau(m_s), \quad r_h = [\![\texttt{e}_1]\!]_{\tau\cup\{v\mapsto s_h\}}(m_s), \{s_h\} \cup s_t = s = [\![\texttt{e}_1]\!]_\tau(m_s); \tag{11}$$

$$[\![\texttt{e}_1\texttt{->sum()}]\!]_\tau = [\![r_h \texttt{ + } r_t\texttt{->sum()}]\!]_\tau; \{r_h\} \cup r_t = [\![\texttt{e}_1]\!]_\tau \tag{12}$$

$$[\![\texttt{e->size()}]\!]_\tau = [\![\texttt{e->collect(1)->sum()}]\!]_\tau \tag{13}$$

$$[\![\texttt{e}_1\texttt{->include(e}_2\texttt{)}]\!]_\tau(m_s) = \begin{cases} \textbf{true} & \textbf{if } \exists r \in r_1 : r = r_2 \\ r_3 & \textbf{if } \exists \mathsf{ite}(r_3, r_2, \_) \in r_1 \\ \textbf{false} & otherwise \end{cases} \quad (r_i = [\![\texttt{e}_i]\!]_\tau(m_s), i \in \{1,2\}) \tag{14}$$

$$[\![\texttt{domain e on p}]\!]_\tau = [\![\texttt{e->exists(x|x=self.p)}]\!]_\tau \tag{15}$$

We use the `MemLimit` constraint at Line 24 in Figure 3 as an example to show how PE works. The constraint is evaluated on the two PM objects, and Figure 5 shows the main steps on `pm1`. The PE starts from `self.hosting`, and is redirected to its derivation (the two OCL constraints are shown in the first two lines of Figure 5). From `allInstances`, the engine obtains a set of three VM objects, and the following `select` transforms it into a set of `if-then-else`. Inside it, `v.plc` calls a configurable property, and is therefore evaluated to a ite. After that, we push the following equation into the ite. As `pm1=pm1` is always **true**, and $\mathsf{ite}(x, true, \bot) = x$, the set is simplifed again. Getting back to the main expression, `collect` substitutes the `mem` value for each VM object, and `sum` joins the three elements by "+", and replace $\bot$ by 0. Finally, we get the inequality as the final output.

```
self.hosting->collect(e|e.mem)->sum() <= self.mem
derive PM.hosting: VM.allInstances()->select(v|v.plc=self)
                                  :
allInstance: {vm1, vm2, vm3}   v.plc: (ite (plc v pm1) pm1 ⊥)   self: pm1
      v.plc=self: {(ite (= (ite (plc v pm1) pm1 ⊥) pm1) vm1 ⊥)... }
                  {(ite (ite (plc v pm1) (= pm1 pm1) ⊥) vm1 ⊥)... }
select: {(ite (plc vm1 pm1) vm1 ⊥), (ite (plc vm2 pm1) vm2 ⊥), (ite (plc vm3 pm1) vm3 ⊥)}
   collect: {(ite (plc vm1 pm1) 6 ⊥), (ite (plc vm1 pm1) 8 ⊥), (ite (plc vm1 pm1) 4 ⊥)}
   sum: (<= (+ (ite (plc vm1 pm1) 6 0) (ite (plc vm2 pm1) 8 0) (ite (plc vm3 pm1) 4 0)) 20)
```

**Fig. 5.** Sample partial evaluation steps

## 5 SMT Solving

Using the generated SMT instance, we leverage an extended constraint solving approach [5] to calculate the appropriate adaptation actions. The generated SMT instance is composed of FOL formulas (SMT constraints), originating from the current context values, configuration values, and adaptation constraints. The latter two categories are *weak constraints*, meaning that they can be violated when necessary. Each weak constraint has a weight generated from a constraint priority or a property cost. The first step is to identify a subset of constraints that we need to remove from the SMT instance in order to make the rest satisfiable. We use a weighted constraint diagnosing approach to find such a subset with the lowest total weight. The second step is to compute the system modifications to satisfy the remaining constraints. Details of this constraint solving approach can be found in our earlier publication [5].

Going back to our running example in Section 2.1, we have shown three potential adaptation solutions, i.e., migrating `vm3` to `pm1`, switching `vm2` and `vm3` (and decrease `pm2.core`), and do nothing. They correspond to three diagnosis: {`BackupSlipt`, `FrequentNear`, `mem-cost`}, {`mem-cost`×2, `core-cost`, `Consolidation`}, {`BackupSlipt`, `FreqentNear`}, respectively, and the second one has the lowest total weight of 120 ($4 \times 10$ and $8 \times 10$ for migrating two VMs, and 20 for decreasing `vm2.core`). The corresponding adaptation solution is `vm1.plc=pm2`, `vm3.plc=pm1`, and `vm1.core=4`.

## 6 Case Study

We apply the approach on the VM placement problem extended from the running example, and use this case study to evaluate: (i) the expressive power of the adaptation modelling language; (ii) the effect of transformation and constraint solving; (iii) the performance improvement achieved by our partial evaluation.

**Implementation.** We implement the adaptation modelling language on the Xtext framework and DresdenOCL toolkit [14], with a text-based syntax and a fully functioning editor (as is shown in Figure 3). We choose Z3 [6] as our constraint solver, and implement the OCL partial evaluation to generate SMT instance in Z3Py, a python-based SMT representation. The generated result is fed into the constraint solving based adaptation engine that we presented in our previous work [5]. The source code is hosted at `github.com/songhui/cspadapt`

**Adaptation modelling.** Our adaptation model on VM placement is based on the general cloud computing concepts from CloudML [15]. We use OCL constraints to model the policies that originate from the following research approaches: 1) Deployment constraints from CloudML, such as 64 bit VM should run on 64 bit PM. 2) Resource limitation and consolidation [7]. 3) Cost of migration [16]. 4) Load balancing between PMs [17], e.g., immigrating VMs out of overloaded PMs, and scatter the VMs with synchronized peak times. 5) Run-time observed logical relations [18], such as frequently communicating VMs should be

placed closely. 6) SLA matching [18], such as VMs with high stability requirement should be placed to specially protected PMs.

All the constraints are specified in a natural and domain specific way, similar to the ones shown in the running example 3. The final adaptation model can be downloaded from `thingml.org/dist/diversify/casestudy.constraint`. The adaptation modelling shows the language's expressive power to specify different adaptation policies, and also reveals a major benefit, i.e., to ease the combination of the policies from different origins.

**Adaptation behaviour.** We test the adaptation model on simulated cloud configurations. From a starting model instance, we randomly generate changes to the system, and feed the changed model to the adaptation engine. The engine outputs the suggested changes, and records the main constraints it followed and discarded when making the decision. Table 2 lists some sample traces. We choose the ones that are from the same starting state (as shown in Figure 6, where `m` and `t` stand for memory and throughput, respectively), and are only involved in 8 particular constraints. When we enlarge `vm2` (see #1 in Table 2), `vm1` is migrated, because it is smaller and therefore cheaper to move. However, when `vm2` exceeds the capacity of `pm1` (#2), itself is migrated. Their destinations are different because of the constraints we listed in the table. When we enlarge `vm4` (#3), the more expensive `vm5` is moved, because moving `vm3` to any PMs would break more expensive constraints. When `vm5` has bigger throughput (#4), `vm4` is moved out to the sole valid destination `pm6`. But since it has synchronised peak time with `vm6`, the latter is moved to `pm2` to avoid `vm4`. When `vm7` has bigger throughput (#5), considering its high stability requirement, the engine moves `vm9` and `vm10` out of the other stable PM to make a room for `vm7`. However, when `vm7`'s stability requirement is lowered down (#6), the engine will move it to a not-so-stable PM, to avoid the cost of moving two VMs. #7 illustrates how we do consolidation: when `v8` is not active, the engine moves it out to free `pm4`, because the cost of immigration is lower than the weight of the consolidation constraint. When `vm6` and `vm9` are observed to be frequently communicating, the engine moves `vm9` to a nearby PM, and brings `vm10` as well. The numbers of constraints in *follows* and *discards* imply the complexity of making the decision.

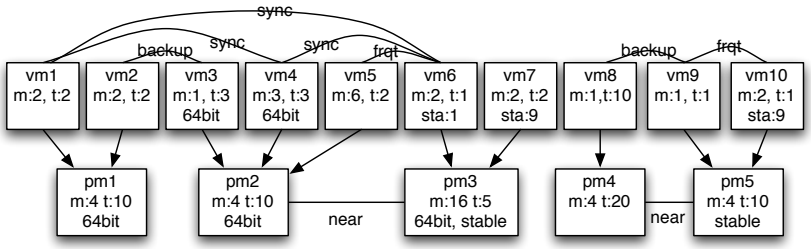


**Fig. 6.** Starting system status of the sample results

**Table 2.** Sample adaptation actions from the VM placement case study.
MC=Migration Cost, SP=Synchronized Peaktime, LB=Load Balance, ML=Memory Limit, BU=Backup, FR=Frequent Communication, ST=Stability, CS=Consolidation, B64=64 bit

| # | change | adaptation | follows | discards |
|---|---|---|---|---|
| #1 | vm2.m:3 | vm1->pm4 | MC SP LB BU | CS |
| #2 | vm2.m:5 | vm2->pm3 | ML BU | MC CS |
| #3 | vm4.m:4 | vm5->pm3 | MC ML FR BU B64 | MC CS |
| #4 | vm5.t:5 | vm4->pm3, vm6->p2 | B64 BU ML FR LB | MC ST CS |
| #5 | vm7.t:6 | vm7->pm5, vm9vm10->pm3 | FR ST LB | MC CS |
| #6 | vm7.t:6, vm7.st:1 | vm7->pm4 | MC LB | ST CS |
| #7 | vm8.t:1 | vm8->pm2 | CS BU | MC |
| #8 | frqt(vm6,vmm9) | vm9->pm2, vm10->pm3 | MC LB FR | MC CS |

**Performance.** The runtime performance of constraint-driven adaptation is acceptable for medium sized systems. We create 6 model instances from the adaptation model on VM placement. In this models, the total number of VMs and PMs are from 15 to 60, and total number of properties are from 140 to 560. For each case, we launch the standard adaptation process for 50 cycles, each started from randomly generated changes (0.5 to 8.5 changes in average), and the average adaptation durations are 0.1, 1.4, 2.2, 7.1, 7.9 and 12.3 seconds, for each model instance. The experiments are performed on a MacBook Pro with Intel i5 CPU and 4G memory. The performance is acceptable since it is still a short time relative to the time it takes to modify topologies and configurations in cloud. Typically migrating one virtual machine in a cloud takes from a half to several minutes. In order to inspect the improvement caused by partial evaluation, we run another 50 cycles for each case from random changes, but with partial evaluation switched off, generating FOL formulas as shown in Figure 4. The adaptation durations are then 0.3, 8.2, 31.5, 49.3, 51.3 are 129.2 seconds, which are significantly longer than the ones with PE. Furthermore, the fraction of reduction increases with the larger models (e.g., the fraction is 3 times faster for the simplest model and about 10 times faster for the largest one)

## 7 Related Work

Research approaches on self-adaptive systems provide many different ways to define adaptation policies. The ECA type of rules are most widely adopted, such as the event triggering in [19] and [20], the guard-action rules in [21], etc. Kephart and Walsh [22] discussed the advantage of declarative policies over imperative ECA rules. Floch et al. [23] utilise declarative properties and utilities functions to capture adaptation policies, but they require predefined system configurations instead of calculate them at runtime. The DiVA project [24] defines a small language to capture constraints as policies, and utilises the Alloy constraint solver to obtain the result. In this work, we support the general purpose OCL language with higher expressive power, and tolerate the conflicts in constraints.

Software engineering techniques, especially model driven engineering, is widely used to tame the complexity of developing adaptive systems. One branch is to

model the system states in high-level architectures, such as in [25] and [26]. But the policies on top of their architectures are essentially ECA rules. Another branch is to support the development process of self-adaptive system. Brun et al. [27] propose a general programming model for self-adaptive systems, based on control loops. Cheng et al. [28] and Baresi et al. [29] use goal-based modelling to help derive adaptation policies from requirements. Such approaches are complementary to the work in this paper, and one of our future plans is to utilise goal models to help identify and organise adaptation constraints from requirements.

The transformation from constraints to SMT instances is related to the approaches that generate constraint satisfiability problems from class diagrams [30,31] or OCL constraints [32], for the purposes of design time verification. When using constraints to guide adaptation at runtime, the searching space for constraint solving is much smaller than at design time, because the context data are known and not changeable. This is the main idea behind our partial evaluation to optimise the generated SMT problem, and differentiate our approach from the existing ones. Partial evaluation [13] is a compiling technique to optimise target code by pre-processing constant values in the source code, and is widely used to support domain specific languages [33]. In our approach, we widen the concept of "constant values" to the current context in an adaptive system.

The modelling process for adaptation constraints is inspired by the construction of domain-specific modelling languages [34,35]: We support domain experts in defining the concepts in a particular application domain, and then the constraints are specified in a domain-specific way, applying these concepts.

## 8 Conclusion

This paper presents a model-driven approach to developing self-adaptative systems. We provide a language for modeling declarative adaptation polices, in the form of domain-specific constraints. Our runtime engine generates an SMT problem from the constraints, optimises it based on the current system state, and calculates the appropriate system reconfigurations. From our previous work, the new contributions in this paper include a modelling language, a new method to encode structural information in object oriented models into SMT, and a new partial evaluation semantics on OCL based on the encoding.

The main limitation of the current approach is the *closed world assumption*. In practice, we loosen this assumption by adding a small number of stub objects before constraint solving, and if any stub object is referred by a real object after the solving, we launch a create-object request to the system. Our future work on this is to generalise this approach and assist developers in customising where a new object is required. The approach can be also used together with an outer adaptation loop that adds or removes VMs, either a manual or an automated one. The case study in the current stage is still a proof of the idea. We will seek for bigger scale applications involving third-party developers, based on our cloud computing research projects.

## References


1. 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)
2. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems (TAAS) **4**(2) (2009) 14
3. Kephart, J.: Research challenges of autonomic computing. In: ICSE, IEEE (2005) 15–22
4. Object Management Group: OMG Object Constraint Language (OCL). http://www.omg.org/spec/OCL/2.3.1/PDF/
5. Song, H., Barrett, S., Clarke, A., Clarke, S.: Self-adaptation with end-user preferences: Using run-time models and constraint solving. In: Model-Driven Engineering Languages and Systems. Springer (2013) 555–571
6. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer (2008) 337–340
7. Hermenier, F., Lorca, X., Menaud, J.M., Muller, G., Lawall, J.: Entropy: a consolidation manager for clusters. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, ACM (2009) 41–50
8. 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
9. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: Computer Aided Verification, Springer (2002) 78–92
10. Guttag, J.V., Horning, J.J.: The algebraic specification of abstract data types. Acta informatica **10**(1) (1978) 27–52
11. Microsoft Research. http://rise4fun.com/z3/tutorial/guide
12. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa (2010) Available at `www.SMT-LIB.org`.
13. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice Hall (New York) (1993)
14. Demuth, B.: The dresden ocl toolkit and its role in information systems development. In: Proc. of the 13th International Conference on Information Systems Development (ISD2004). (2004)
15. Ferry, N., Rossini, A., Chauvel, F., Morin, B., Solberg, A.: Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In: CLOUD 2013: IEEE 6th International Conference on Cloud Computing. (2013) 887–894
16. Meng, X., Pappas, V., Zhang, L.: Improving the scalability of data center networks with traffic-aware virtual machine placement. In: INFOCOM, 2010 Proceedings IEEE, IEEE (2010) 1–9
17. Bobroff, N., Kochut, A., Beaty, K.: Dynamic placement of virtual machines for managing sla violations. In: Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on, IEEE (2007) 119–128
18. Zhang, X., Zhang, Y., Chen, X., Liu, K., Huang, G., Zhan, J.: A relationship-based vm placement framework of cloud environment. In: Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference, IEEE Computer Society (2013) 124–133

19. Keeney, J., Cahill, V.: Chisel: A policy-driven, context-aware, dynamic adaptation framework. In: Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on, IEEE (2003) 3–14
20. Kephart, J.O., Das, R.: Achieving self-management via utility functions. Internet Computing, IEEE **11**(1) (2007) 40–48
21. David, P.C., Ledoux, T., et al.: Safe dynamic reconfigurations of fractal architectures with fscript. In: Proceeding of Fractal CBSE Workshop, ECOOP. Volume 6. (2006)
22. 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
23. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using architecture models for runtime adaptability. Software, IEEE **23**(2) (2006) 62–70
24. Morin, B., Barais, O., Jezequel, J., Fleurey, F., Solberg, A.: Models@ run. time to support dynamic adaptation. Computer **42**(10) (2009) 44–51
25. Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer **37**(10) (2004) 46–54
26. Sicard, S., Boyer, F., De Palma, N.: Using components for architecture-based management: the self-repair case. In: ICSE, ACM (2008) 101–110
27. Brun, Y., Serugendo, G.D.M., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Engineering self-adaptive systems through feedback loops. In: Software Engineering for Self-Adaptive Systems. Springer (2009) 48–70
28. 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
29. Baresi, L., Pasquale, L., Spoletini, P.: Fuzzy goals for requirements-driven adaptation. In: RE, IEEE (2010) 125–134
30. Maoz, S., Ringert, J., Rumpe, B.: CD2Alloy: Class diagrams analysis using alloy revisited. MODELS (2011) 592–607
31. 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
32. 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
33. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM computing surveys (CSUR) **37**(4) (2005) 316–344
34. Kelly, S., Tolvanen, J.P.: Domain-specific modeling: enabling full code generation. John Wiley & Sons (2008)
35. 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