Workflow Variability for Autonomic IoT Systems

Workflow Variability for Autonomic IoT Systems

Damian Arellanes and Kung-Kiu Lau
School of Computer Science
The University of Manchester
Manchester M13 9PL, United Kingdom
{damian.arellanesmolina, kung-kiu.lau}@manchester.ac.uk

***Abstract*—Autonomic IoT systems require variable behaviour at runtime to adapt to different system contexts. Building suitable models that span both design-time and runtime is thus essential for such systems. However, existing approaches separate the variability model from the behavioural model, leading to synchronization issues such as the need for dynamic reconfiguration and dependency management. Some approaches define a fixed number of behaviour variants and are therefore unsuitable for highly variable contexts. This paper extends the semantics of the DX-MAN service model so as to combine variability with behaviour. The model allows the design of composite services that define an infinite number of workflow variants which can be chosen at runtime without any reconfiguration mechanism. We describe the autonomic capabilities of our model by using a case study in the domain of smart homes.**

***Index Terms*—Internet of Things, autonomic systems, DX-MAN, exogenous connectors, algebraic service composition, workflow variability, models@runtime, smart home**

I. Introduction

The Internet of Things is an emerging paradigm that envisions the interconnection of everything through novel distributed services which are combined into complex workflows using service composition mechanisms. Workflows represent IoT systems composed of billions of services with an overwhelming number of interactions. Thus, it becomes infeasible to manually manage such systems as the scale and complexity increases.

Autonomicity is a crucial desideratum for the management of complex large-scale IoT systems operating in highly dynamic environments. It is a property that allows adapting behaviour at runtime to different contexts with minimal or no human intervention. Autonomicity thus requires workflow variability for the definition of alternative system behaviours.

Although relatively trivial in static IoT systems, changing behaviour at runtime in highly variable environments is a complex and challenging task. For that reason, variability-based autonomicity has been an active research topic for software engineering in the last decade [1], [2]. Although there are many proposals for managing variability, they fail at incorporating variability in behavioural elements (i.e., in the solution space) while avoiding the cumbersome time-consuming task of dynamic reconfiguration [1], [3].

This paper extends the semantics of the DX-MAN service model [4], [5], [6] with autonomicity capabilities for IoT systems. The semantics allows adapting workflows at runtime to different contexts without requiring any dynamic reconfiguration mechanism. Our contribution is thus two-fold: (i) *a model* that combines variability with behaviour in the solution space, while providing an infinite number of workflow variants for composite IoT services; and (ii) *an approach* that avoids dynamic reconfiguration (by using *non-deployable* and *executable only* workflows).

The rest of the paper is structured as follows. Sect. II describes the main constructs of the DX-MAN model. Sect. III presents the mechanism to realize workflow variability. Sect. IV describes the autonomicity dimension of the model. Sect. V presents a case study to show autonomicity in a case study. Sect. VI describes the related work. Finally, Sect. VII presents the conclusions and the future work.

II. DX-MAN Model

DX-MAN is an algebraic model for IoT systems where services and exogenous connectors are first-class entities. An exogenous connector is a deployable entity that executes multiple workflows with explicit control flow. A service S is a stateless distributed software unit with a well defined interface, which can be either atomic (A) or composite (C):

$$S := A|C \tag{1}$$

A service defines a workflow space W which is a non-empty (finite or infinite) set, where each $w \in W$ is a workflow variant that represents an alternative service behaviour. The workflow space constitutes the service interface, and is semantically equivalent to a service S:

$$S \equiv W = \{w_1, w_2, \ldots\} \tag{2}$$

A. Atomic Services

An atomic service A is a tuple $\langle IC, O\rangle$ consisting of an invocation connector IC and a non-empty finite set O of j primitive operations (Fig. 1). It is formed by connecting an invocation connector with a computation unit.

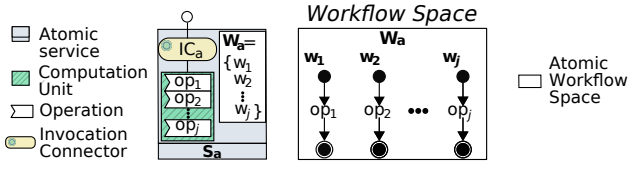

Fig. 1. A DX-MAN atomic service defines j workflows: $|W| = j$.

A computation unit is not allowed to call other computation units, and is the place where j service operations are implemented using well-known technologies such as REST. To satisfy

an external request, an invocation connector is responsible for executing a workflow in W.

Fig. 1 shows that an atomic service $S_a \in A$ defines an atomic workflow space W_a s.t. $|W_a| = j$ and each $w_{i\in[1,j]} \in W_a$ is a workflow invoking an operation $op_{i\in[1,j]} \in O$. The atomic workflow space W_a is the interface of S_a.

B. Algebraic Composition

Our notion of algebraic service composition is inspired by algebra where functions are hierarchically composed into a new function of the same type. The resulting function can be further composed with other functions, yielding a more complex one. Algebraic service composition is then the operation by which a composition connector composes k services into a more complex service. The result is a (hierarchical) composite service whose interface is constructed from the sub-service interfaces. Formally, a composite service is a tuple $\langle CC, \mathcal{W}\rangle$ consisting of:

- a composition connector CC that invokes multiple workflows defined by the composite service, and
- a non-empty finite $\mathcal{W}$ set which is a family of non-empty (finite of infinite) sets of sub-workflow spaces s.t. each $W_i \in \mathcal{W}, i = 1, \ldots, k$ is a workflow space of either an atomic sub-service or a composite sub-service.

A composite service is a variation point which defines a new non-empty (finite or infinite) workflow space W using the sub-workflow spaces $\mathcal{W}$ via algebraic references (Fig. 2). W serves as the composite service interface, and is available to more complex composites.

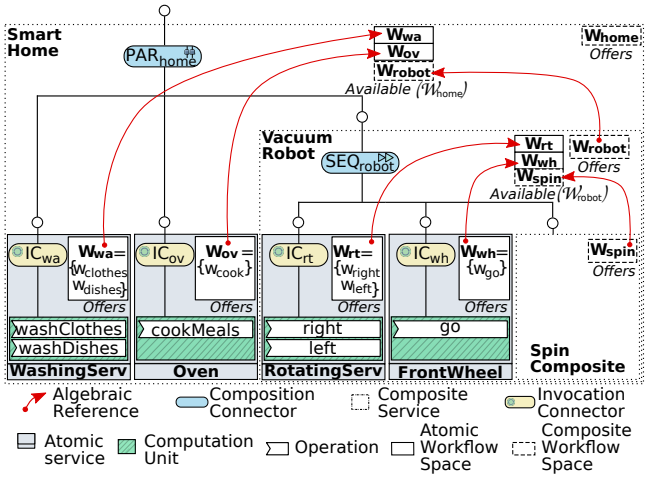

Fig. 2. Algebraic Composition for a Smart Home.

Fig. 2 depicts a two-level DX-MAN composition for a smart home with four atomic services (i.e., *WashingServ*, *Oven*, *RotatingServ* and *FrontWheel*) and three composite services (i.e., *SpinComposite*, *VacuumRobot* and *SmartHome*). The services are described in Sect. III. For the sake of clarity, we omit the internal structure of *SpinComposite*, but we show its interface: the composite workflow space W_{spin}. The interfaces of *WashingServ*, *Oven*, *RotatingServ* and *FrontWheel* are the atomic workflow spaces $W_{wa} = \{w_{clothes}, w_{dishes}\}$, $W_{ov} = \{w_{cook}\}$, $W_{rt} = \{w_{right}, w_{left}\}$ and $W_{wh} = \{w_{go}\}$, respectively. The services *RotatingServ*, *FrontWheel* and *SpinComposite* are composed into *VacuumRobot* (using the composition connector SEQ_{robot}, see Fig. 3). Thus, the interfaces W_{rt}, W_{wh} and W_{spin} are available in *VacuumRobot* which, in turn, defines the composite workflow space W_{robot}. Then, *WashingServ*, *Oven* and *VacuumRobot* are composed into the top-level composite *SmartHome* (using the composition connector PAR_{home}, see Fig. 6). So, *SmartHome* has available the interfaces W_{wa}, W_{ov} and W_{robot}, and yields the composite workflow space W_{home}.

C. Workflow Selection

A composition connector CC is a variability operator that defines the alternative behaviours of a composite service. It is a function that defines a workflow space W, given a family of sub-workflow spaces $\mathcal{W}$:

$$CC : \mathcal{W} \mapsto W \tag{3}$$

A composition connector has access to atomic sub-workflow spaces, but not to composite sub-workflow spaces. This is because a composite sub-service is a black box whose behaviour is unknown. Hence, a composition connector operates on n elements to define sequential, branching or parallel workflows for a composite $c \in C$. The total number of elements n is the sum of the cardinality of atomic sub-workflow spaces and the number of composite sub-services:

$$n = \sum_{i=1}^{|\mathcal{W}_c|} \begin{cases} |W_c^i| & s_c^i \in A \\ 1 & s_c^i \in C \end{cases} \tag{4}$$

where $\mathcal{W}_c \in \mathcal{W}$ is the set of sub-workflow spaces of the composite c, $n \geq |\mathcal{W}_c|$ and $W_c^i \in \mathcal{W}_c$ is the workflow space of a sub-service S_c^i.

At design-time, an *abstract workflow tree* is automatically created for a composite service, as a result of composition. It represents the hierarchical control flow structure of a composite service, where n leaves are atomic workflows, composite workflow spaces or any combination thereof (e.g., Fig. 3). The leaves are also referred to as the *elements* of a workflow tree. The edges represent customizable control flow parameters (e.g., execution order or conditions) which are determined by the composition connector being used. In our current implementation, abstract workflow trees are JSON objects.

A *concrete workflow tree* enables the selection of a workflow variant at runtime. It particularly sets specific values for the customizable control flow parameters of an abstract workflow tree, in order to select the elements (i.e., atomic workflows or composite workflow spaces) to include in a workflow out of n possibilities (e.g., Fig. 4). In our current implementation, concrete workflow trees are also JSON objects.

III. Composition Connectors as Variability Operators

This section describes some of the composition connectors currently supported by DX-MAN, namely sequencer, parallelizer and exclusive selector. Although the inclusive selector is also supported, we do not describe it due to space constraints.

A. Sequencer

A *sequencer* connector SEQ uses the Kleene star operation to allow the repetition of n elements, resulting in infinite sequences. It then defines an infinite workflow space for a composite service s.t. each $w_i \in W, i = 1, \ldots, \infty$ is a sequential workflow. A sequencer is a function defined as:

$$SEQ : \mathcal{W} \mapsto W \tag{5}$$

where $|W| = \infty$.

1) Example: Consider a vacuum robot that cleans a room in a smart home using a composite service *VacuumRobot*. It relies on two atomic services and one composite service to navigate efficiently, as shown by Fig. 3. The atomic service *RotatingServ* provides two operations for turning the robot to the *left* and *right*, respectively. The atomic service *FrontWheel* offers the operation *go* to move the robot one unit forward. There is also a *SpinComposite* service that enables the robot to spin 360°, in order to clean the dirtiest areas of the room. For clarity, we do not show the internal structure of *SpinComposite*.

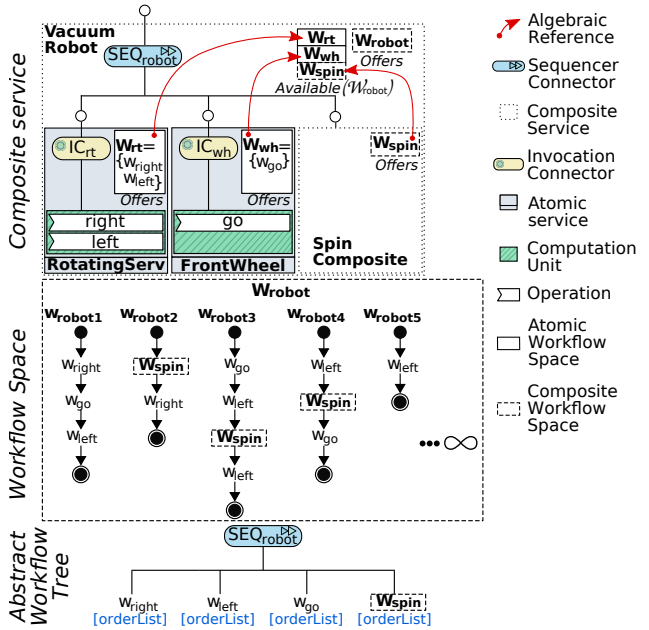

Fig. 3. A sequencer defines ∞ workflows for a composite service: $|W| = \infty$. In this example, there are ∞ sequential workflows for *Vacuum Robot*.

The sequencer connector SEQ_{robot} composes the services *RotatingService*, *FrontWheel* and *SpinComposite* into *VacuumRobot*, resulting in the infinite workflow space W_{robot}. Fig. 3 illustrates a few workflow variants for *VacuumRobot*. For instance, the variant w_{robot4} indicates that the atomic workflow w_{left} is executed before the composite workflow space W_{spin} which, in turn, is executed before the atomic workflow w_{go}. Note that W_{spin} cannot be accessed by the *VacuumRobot* since the *SpinComposite* sub-service is a black box entity which can take any possible behaviour. Instead, only atomic workflow spaces (i.e., W_{rt} and W_{wh}) can be accessed.

2) Workflow Selection: An abstract workflow tree of a sequencer requires the specification of the execution order for n elements. An execution order is a non-negative integer that reflects the position of an element in a workflow. As a sequencer allows repetition, an element requires an order list $[order_1, order_2, \ldots]$, as shown by Figs. 4 and 5. Elements with no order lists are not included in a workflow and, to ensure consistent sequences, an order cannot appear in multiple lists.

Fig. 4 shows an example of a concrete workflow tree for choosing the sequential workflow w_{robot3} for the composite *VacuumRobot*. The element w_{right} is left out as it does not have any order list. Fig. 5 illustrates another example for the selection of the sequential workflow w_{robot1} which now excludes the composite workflow space W_{spin}.

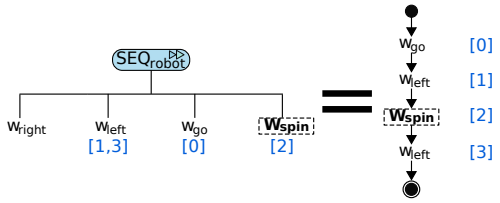

Fig. 4. Concrete workflow tree for choosing the sequential workflow w_{robot3} for the $VacuumRobot$ composite.

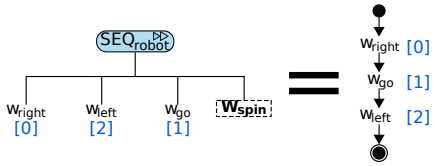

Fig. 5. Concrete workflow tree for choosing the sequential workflow w_{robot1} for the $VacuumRobot$ composite.

B. Parallelizer

A *parallelizer* connector PAR allows the execution of multiple elements in parallel. As it supports element repetition, it defines ∞ parallel workflows for a composite service s.t. each $w_i \in W, i = 1, \ldots, \infty$ is a workflow executing all the elements in parallel. Formally, a parallelizer is a function defined as:

$$PAR : \mathcal{W} \mapsto W \tag{6}$$

where $|W| = \infty$.

1) Example: Consider the composition depicted in Fig. 6 where *SmartHome* is the top-level composite which is able to do the chores for a user. The atomic service *WashingServ* provides the operations *washClothes* and *washDishes* for washing clothes and washing dishes, respectively. The atomic service *Oven* offers the operation *cookMeals* for cooking breakfast, lunch and dinner in a specific day. The composite service *VacuumRobot*, previously presented in Fig. 3, is also available for the smart home. For clarity concerns, we omit the internal structure of *VacuumRobot* and we only show the respective interface.

A parallelizer connector PAR_{home} composes *WashingServ*, *Oven* and *VacuumRobot* into *SmartHome*, resulting in the

workflow space W_{home} of infinite parallel workflows. Some workflow variants are displayed in Fig. 6. For instance, the variant w_{home2} executes the atomic workflows $w_{clothes}$ and w_{cook} in parallel. w_{home4} is another variant that leverages the support for repetition so as to execute the atomic workflow w_{cook} in three different tasks. This is useful for cooking three meals for three different people simultaneously.

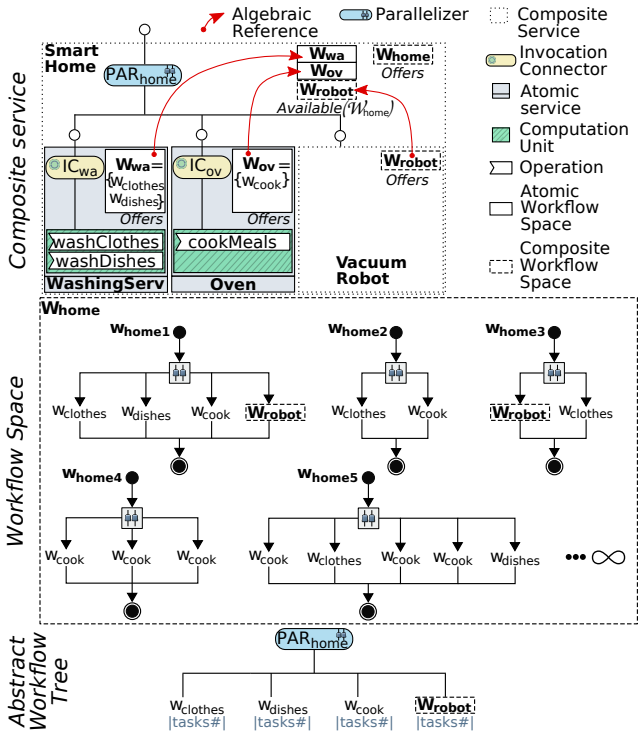

Fig. 6. A parallelizer defines ∞ workflows for a composite service: $|W| = \infty$. In this example, there are ∞ parallel workflows for *SmartHome*.

2) Workflow Selection: The abstract workflow tree of a parallelizer allows the selection of elements to include in a parallel workflow, and there are n elements that can be selected with repetition allowed. Each element requires the specification of a natural number that represents the number of tasks for that particular element, and elements with no tasks are excluded from the workflow being constructed. A task basically represents the number of times an element is repeated in a parallel workflow. So, at runtime it is an invocation thread.

Fig. 7 shows a concrete workflow tree for choosing the variant w_{home5}. It defines three tasks for the atomic workflow w_{cook}, one task for the atomic workflow $w_{clothes}$ and another one for the atomic workflow w_{dishes}. This means that the smart home washes dishes, prepares three meals and washes clothes at the same time. The composite workflow space W_{robot} is excluded from w_{home5}. Fig. 8 shows another concrete workflow tree for choosing w_{home3} which only includes the composite workflow space W_{robot} and the atomic workflow $w_{clothes}$.

C. Exclusive Selector

An *exclusive selector* $XSEL$ defines a workflow space with $2^n - 1$ exclusive branching workflows for a composite service.

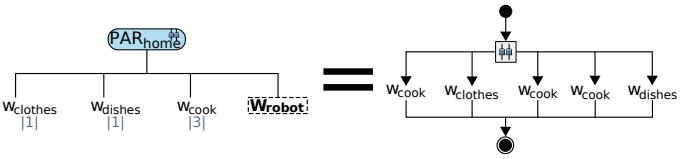

Fig. 7. Concrete workflow tree for choosing the parallel workflow w_{home5} for the $SmartHome$ composite.

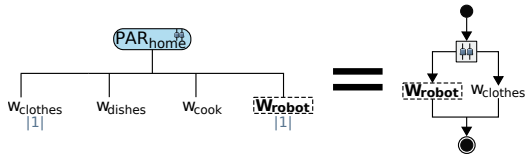

Fig. 8. Concrete workflow tree for choosing the parallel workflow w_{home3} for the $SmartHome$ composite.

Each workflow $w_i \in W, i = 1, \ldots, (2^n - 1)$ contains at least one element out of n possibilities, and chooses a single element to be executed. An exclusive selector is a function defined as:

$$XSEL : \mathcal{W} \mapsto W \tag{7}$$

where $|W| = 2^n - 1$.

1) Example: Consider a speaker controlled by a composite service *Player* for playing audio in a room. It has an atomic service *Music* that provides two operations for playing Jazz and playing pop music, respectively. There is also an atomic service *News* for reading the most recent news, and a composite service *WeatherReport* for listening to the weather forecast. For clarity, we omit the internal structure of *WeatherReport*.

Fig. 9 shows that the exclusive selector $XSEL_{play}$ composes the services *Music*, *News* and *WeatherReport* into *Player*. The composition process results in the workflow space W_{play} of $2^4 - 1 = 15$ exclusive branching workflows, as there are four elements available: the atomic workflows w_{jazz}, w_{pop} and w_{news}, and the composite workflow space $W_{weather}$. Fig. 9 illustrates some workflow variants for the composite *Player*. For instance, the workflow w_{play15} may execute w_{jazz}, w_{pop} or $W_{weather}$. Another variant is w_{play6} which chooses to play either jazz or pop.

2) Workflow Selection: The abstract workflow tree of an exclusive selector chooses the elements to include in a workflow out of n possibilities. To do so, a binary tag must be specified for each element, so elements tagged with *One* are included, whilst elements tagged with *Zero* are not included. A single condition must be specified for the entire branch because an exclusive selector applies 1 condition to multiple elements, thereby choosing only one element at a time. Thus, the maximum number of possible executions is the same number of elements included in the workflow, plus an empty execution. The empty execution means that no element is executed when the condition holds false at runtime. In our current implementation, we use Java interfaces for defining conditions.

Fig. 10 shows a concrete workflow tree for choosing the variant w_{play15} which excludes the atomic workflow w_{news}. It applies a single condition to w_{jazz}, w_{pop} and $W_{weather}$ for

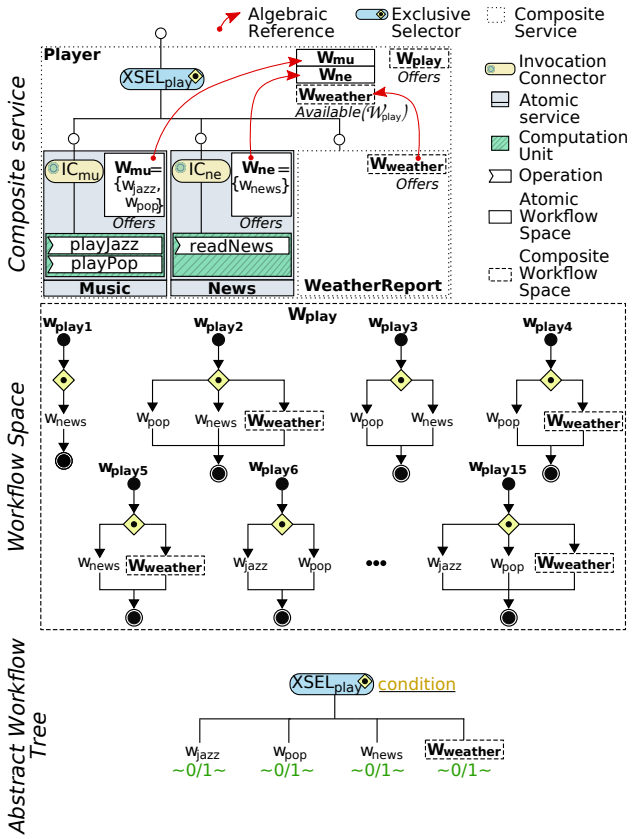

Fig. 9. An exclusive selector defines $2^n - 1$ workflows for a composite service: $|W| = 2^n - 1$. In this example, there are $2^4 - 1 = 15$ exclusive branching workflows for *Player*.

playing Jazz at nights, pop music on afternoons or the *weather forecast* in the morning. w_{play15} has four possible executions.

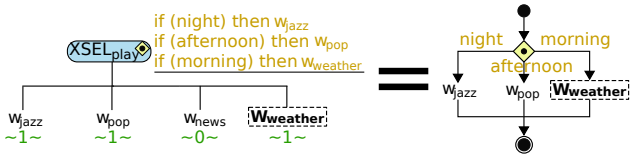

Fig. 10. Concrete workflow tree for choosing the exclusive branching workflow w_{play15} for the *Player* composite.

Fig. 11 illustrates another concrete workflow tree for choosing the workflow variant w_{play3}. It has a condition for playing pop music if there are multiple users present, or listening to the news when there is only one user. As it uses an *if-else* condition, w_{play3} enables only two possible executions.

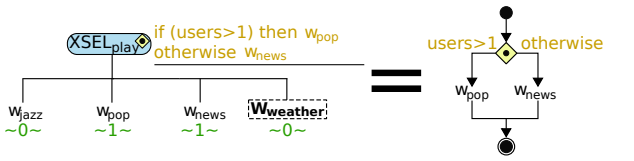

Fig. 11. Concrete workflow tree for choosing the exclusive branching workflow w_{play3} for the *Player* composite.

IV. Emergent Behaviour of DX-MAN Compositions using Feedback Control Loops

This section describes the mechanism that enables an autonomous selection of workflow variants at runtime in composite services.

In DX-MAN, workflow spaces represent the adaptation space of a composite service, since they provide a wide range of workflow variants, each representing a different behaviour. Unlike existing approaches, DX-MAN does not require to link the variability model with the behavioural model, as those dimensions are mixed in the semantics of a composite service.

The selection of workflow variants (i.e., changing behaviour) takes place at runtime whenever the context changes. This is done by building the concrete workflow tree that best adapts to the current context. For this, we use Monitoring, Analysis, Planning, Execution and Knowledge (MAPE-K) [7] which endow composite services with autonomicity. MAPE-K is a feedback control loop consisting of multiple sensors, a monitor, an analyzer, a planner, an executor, an effector and a knowledge base. Fig. 12 shows that a MAPE-K loop manages a composite service and collects information from the external context (e.g., the surrounding environment or user preferences). Remarkably, autonomicity is an orthogonal dimension to control, data and computation in the DX-MAN model.

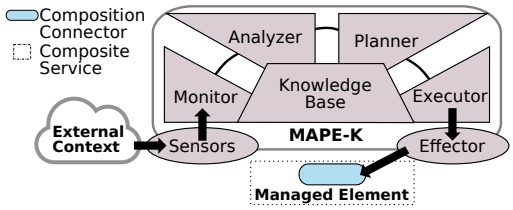

Fig. 12. MAPE-K for DX-MAN.

The MAPE-K components are able to read and update the *knowledge base* which stores relevant information for realizing autonomic behaviour. By default, the knowledge base stores the abstract workflow tree for the managed composite service.

The *monitor* uses sensor data to build a context model for the external environment, which is used by the analyzer to decide if a new behaviour is required. If so, the planner determines the best workflow variant for the current context state, resulting in a plan that is passed to the *executor* which transforms it into a concrete workflow tree matching the structure of the abstract workflow tree. Finally, the executor uses the effector to change the behaviour of the managed composite service, by executing the chosen concrete workflow tree. In our current implementation, the context model, the context state, plans and workflow trees are JSON documents. We do not show the source code due to space constraints, but JSON samples are available at https://gitlab.cs.man.ac.uk/mbaxrda2/dxman.

At runtime, control blocks when it reaches a composition connector. Once a MAPE-K determines the "best" workflow for a managed composite service, the executor resumes the workflow execution by passing a concrete workflow tree to the connector of the managed composite.

As every composite service is managed by a different MAPE-K loop, any composite at any level in the hierarchy is able to change its behaviour at runtime independently. This inevitably requires ensuring consistency for the current workflow execution. Fortunately, dynamic workflow deployment is not required since DX-MAN workflows are executable only. Whenever a new workflow is required, the effector kills the thread of the current workflow execution, thereby instantly stopping the sub-workflows being executed by the managed composite. A new thread is then created for the execution of the new workflow.

Workflow selection may potentially happen simultaneously at multiple levels in the hierarchy. So, continuously changing sub-workflows leads to an emergent behaviour of the whole system. MAPE-K loops are continuously operating, even though control flow has not yet reached the managed composition connector. However, they can only change the composite service behaviour, by executing a concrete workflow tree, when control flow has passed through or is blocked in the managed connector.

A running IoT system is practically a complex workflow consisting of sub-workflows s.t. each sub-workflow represents a composite service behaviour. This is precisely due to the hierarchical structure of a DX-MAN composition. By contrast, MAPE-K loops are not structured hierarchically as they never interact. Instead, they only select a workflow for the managed composite service (at any level in the hierarchy) and they execute new workflows (when control is blocked in the managed composition connector) or replace an existing workflow with a "better one" (when control has already passed through).

V. Case Study: Smart Home

This section presents a case study in the domain of end-user smart homes where the external context (e.g., user presence) is always changing and users are always willing a quick workflow selection. So, existing approaches for variability-based autonomicity (see Sec. VI) are not suitable for smart homes. This is because those approaches require time for changing behaviour due to dynamic reconfiguration and/or provide a limited number of variants which may not be suitable for some contexts. We leverage the capabilities of DX-MAN to avoid dynamic reconfiguration and provide a wide range of workflow variants. The DX-MAN composition for our case study is basically the composite service *SmartHome* described in Sect. II and depicted in Fig. 2. Although we endow every composite service with its own MAPE-K loop, this section just focuses on the autonomicity of *VacuumRobot* and *SmartHome*.

A. Autonomic Vacuum Robot Composite

The goal of the *VacuumRobot* composite (Fig. 3) is to clean a room as efficiently as possible by continuously changing the robot trajectory. As it operates on a dynamic environment where people is always moving, the robot changes trajectory whenever an obstacle is detected. For that, a MAPE-K loop chooses the most efficient trajectory (i.e., the best sequential workflow) that cleans every accessible areas of the room while avoiding collisions.

The MAPE-K is equipped with three range sensors that perceive the external environment of the vacuum robot. The infrared proximity sensor is used for detecting obstacles while the robot moves around. A cliff sensor is important to avoid driving over cliffs (e.g., stairwells or ledges) and a dirt sensor detects the dirtiness level on the current position of the robot.

The MAPE-K *knowledge* contains information about the surrounding map, in addition to the abstract workflow selection tree of *VacuumRobot*. The map contains information about obstacles and dirtiness levels in the room which are updated by the *monitor* to improve future navigation, and is queried when a new trajectory is required. We assume that the dirtiness levels are determined by any existing approach (e.g., Poisson processes [8]). We also assume that the map is bidimensional where each position is a disk shape fitting the robot size, as shown in [9]. In particular, a disk can be either an obstacle or a free space with a (high or normal) dirtiness level.

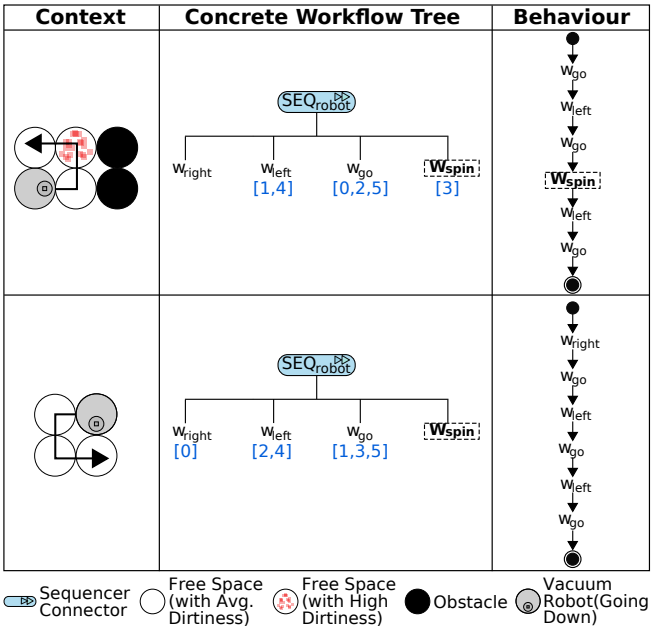

Fig. 13. Possible behaviours for the *VacuumRobot* composite.

The *analyzer* determines if there is an obstacle in the current robot position, and discovers new areas to cover. The *planner* is notified when the analyzer detects an obstacle, and uses *scan matching online cell decomposition* [10] for finding the best trajectory. To ensure a harder cleaning, we modified such an approach so as to enable trajectories where the robot spins on the dirtiest areas of the room. This mechanism is out of the scope of this paper.

The *executor* transforms the best trajectory into a sequence of actions (i.e., the best sequential workflow variant) the robot needs to carry out for the current context. For this, it uses the abstract workflow tree to build a concrete workflow tree, and then triggers the effector. Finally, the effector executes the variant by passing the concrete workflow tree to the sequencer SEQ_{robot}. The current execution (if any) is overridden (i.e., stopped and replaced) by the new workflow variant. Fig. 13

shows two possible behaviours for the *VacuumRobot* composite in two different contexts. Due to space constraints, the contexts are fragments of the map presented in [9].

B. Autonomic Manager for the Smart Home Composite

The *SmartHome* composite does chores in parallel for a user, while minimizing energy consumption and maximizing tidiness. Its behaviour changes once a day and depends on user preferences, changes in the external environment, and non-functional properties of *SmartHome* elements. Table I shows the annotated non-functional properties for $w_{clothes}$, w_{dishes}, w_{cook} and W_{robot}. The *userPresence* property takes a binary value to indicate whether the element should be executed when the user is at home (i.e., *One*) or away (i.e., *Zero*). The *energy* property defines the average discrete amount of energy (in Watts per hour) required for the execution of an element. The *tidiness* property determines the discrete level of tidiness resulting from the execution of a specific element. The sum of all *tidiness* values must be equal to *One*. It is also important to note that the non-functional properties we assume can be much more complex in other case studies.

Element	UserPresence(u)	Energy(e)	Tidiness(t)
$w_{clothes}$	0	500.0	0.25
w_{dishes}	0	350.0	0.25
w_{cook}	1	1300.0	0.10
W_{robot}	0	150.0	0.40

TABLE I
NON-FUNCTIONAL PROPERTIES FOR THE ELEMENTS OF *SmartHome*.

The *userPresence* values depend on user-defined rules which indicate to hoover and wash when the user is away, in order to avoid accidents and noise disturbances. Thus, only w_{cook} has a *userpresence* of 1.

A workflow variant $w_i \in W_{home}$ includes v elements s.t. $v \leq n$, and its properties are computed using Equations 8, 9 and 10. The *userPresence* $u(w_i)$ is an average s.t. each $u_i^x, x = 1, \ldots, v$ is the *userPresence* value of an element x of w_i. The *energy* consumption $e(w_i)$ is a sum s.t. each $e_i^x, x = 1, \ldots, v$ is the *energy* consumption of an element x of w_i. Similarly, the level of *tidiness* $t(w_i)$ is a sum s.t. each $t_i^x, x = 1, \ldots, v$ is the *tidiness* value of an element x of w_i. Thus, the workflow variant w_i with all the elements of *SmartHome* (i.e., $v = n$), provides the highest tidiness and the highest energy consumption.

$$u(w_i) = \frac{\sum_{x=1}^{v} u_i^x}{v} \tag{8}$$

$$e(w_i) = \sum_{x=1}^{v} e_i^x \tag{9}$$

$$t(w_i) = \sum_{x=1}^{v} t_i^x \tag{10}$$

The external context ϕ changes daily and is modeled by setting the user presence $u(\phi)$, the current energy cost $c(\phi)$ (in dollars per Watt-hour) and a threshold $\tau(\phi)$ which defines the maximum amount (in dollars) the user is willing to spend for energy (in a given day). We particularly define utility functions to express the quantitative level of satisfaction of workflow variants for the current context [11]. Overall, the objective is to minimize energy cost and maximize tidiness. The utility functions range from [0,1] where 0 reflects the worst satisfability and 1 means the opposite.

Equation 11 is the utility function f_1 that computes the suitability of a workflow variant $w_i \in W_{home}$ for the user presence. Equation 11 describes a piecewise utility function f_2 that determines how well w_i minimizes energy costs. Finally, Equation 13 is the utility function f_3 that computes the contribution to tidiness of w_i.

$$f_1(w_i, \phi) = 1 - \mid u(\phi) - u(w_i) \mid \tag{11}$$

$$f_2(w_i, \phi) = \begin{cases} 1 - \frac{e(w_i) \cdot c(\phi)}{\tau(\phi)} & e(w_i) \cdot c(\phi) < \tau(\phi) \\ 0 & e(w_i) \cdot c(\phi) \geq \tau(\phi) \end{cases} \tag{12}$$

$$f_3(w_i) = t(w_i) \tag{13}$$

Equation 14 computes the overall utility $U(w_i, \phi)$ of a workflow variant $w_i \in W_{home}$ for the current context ϕ. The weights ω_1, ω_2 and ω_3 define the preference of taking into account user presence, the priority of considering the energy cost and the preference of having a tidy environment, respectively. They are continuous values in the range $[0, 1]$ s.t. a higher value indicates a higher preference. For our experiments, $\omega_1 = \omega_2 = \omega_3 = 1$.

$$U(w_i, \phi) = \frac{\omega_1 \cdot f_1(w_i, \phi) + \omega_2 \cdot f_2(w_i, \phi) + \omega_3 \cdot f_3(w_i)}{\omega_1 + \omega_2 + \omega_3} \tag{14}$$

The behaviour of the *SmartHome* composite is controlled by a MAPE-K loop which has three sensors collecting information from the external context ϕ, namely user presence, current energy costs (from the energy supplier) and a threshold value (continuously changed by the user). In addition to the abstract workflow tree of *SmartHome*, the *knowledge base* includes the aforementioned utility functions, as well as context values and selected workflows from previous days. It also contains the values of the non-functional properties presented in Table I.

The *monitor* is executed once a day, and builds a relationship between context properties and sensor values. Some examples of context models are presented in Table II. The *analyzer* receives a context model as an event, and triggers an Event-Condition-Action (ECA) rule. The rule decides a new plan is required if the current context values are different from the previous day; otherwise, it executes the plan from the previous day and no planning phase is performed.

As the size of W_{home} is infinite (Fig. 6), evaluating all workflow variants is infeasible. For that reason, we propose a *planner* using a metaheuristic approach which finds the most suitable workflow for a specific context. For clarity, we reduce the space search by omitting element repetition for every $w_i \in W_{home}$. So, elements of selected workflow variants have

Day (ϕ)	UserPresence(u_ϕ)	EnergyCost(e_ϕ)	Threshold(τ_ϕ)
1	0	0.00014	0.2
2	1	0.00007	0.6
3	1	0.00012	0.3
4	0	0.00013	0.5

TABLE II
POSSIBLE CONTEXT MODELS.

only one task. As *SmartHome* has four elements (i.e., $w_{clothes}$, w_{dishes}, w_{cook} and W_{robot}), there would be $2^4 - 1 = 15$ workflow variants in W_{home}. Although $|W_{home}|$ is relatively small, we use a genetic algorithm to show what a planner would do for larger workflow spaces.

A chromosome represents a workflow variant with four boolean genes.[1] Fig. 14 shows that the order of genes is mandatory as each gene represents an element of the *SmartHome* composite, where a gene $Zero$ means that the element is not selected, whilst a gene One entails that the element has one task. For instance, the chromosome *0101* represents a workflow variant for executing w_{dishes} and W_{robot} in parallel. A population is thus a set of workflow variants representing possible solutions for the current context ϕ. Each variant is evaluated by the utility function presented in Equation 14.

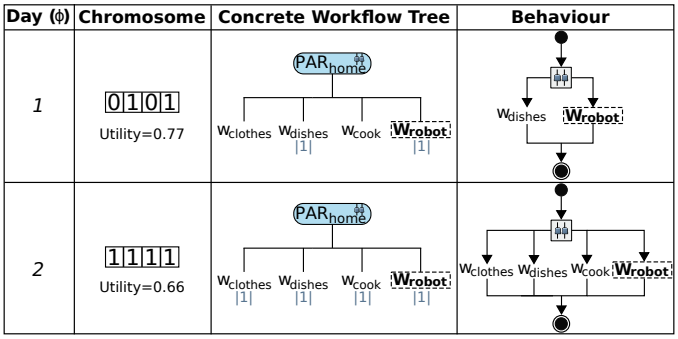

Fig. 14. Possible behaviours for the *SmartHome* composite.

After two workflow variants are selected in a generation, a one-point crossover operator is used. The crossover point is randomly selected and replaces the gene of one variant with the gene of another one. The result is two children representing two new workflow variants for the next generation. To increase diversity, we introduce mutation by randomly selecting a gene and flipping it from zero to one, or viceversa. For our implementation, we use the NSGA-II algorithm and the MOEA framework. Our source code is available at https://gitlab.cs.man.ac.uk/mbaxrda2/dxman. As this is a relatively small problem, the parameters of the genetic algorithm are as follows: population size is 8, crossover probability is 0.5, mutation probability is 0.2 and number of iterations is 20.

The result of the planner is a chromosome representing the optimal parallel workflow for the current context. The *executor* then creates a concrete workflow tree that fits the plan. Fig. 14 shows the behaviours of *SmartHome* for adapting to the context of days 1 and 2 (described in Table II). We only show two behaviours due to space constraints. To change the behaviour of the *SmartHome* composite, the *effector* passes the respective concrete workflow tree to the parallelizer PAR_{home} at runtime.

[1]For infinite workflow spaces, we could consider a chromosome where each gene is a non-negative integer in $[0, \infty]$.

VI. RELATED WORK

The related work is classified into two categories concerning workflow variability: *solution space variability* and *Models@Runtime*. We omit approaches using variability at the planning-level (e.g., [12]) as they do not propose any model constructs for supporting workflow variability, but they are built on top of existing component models with reconfiguration capabilities (e.g., Fractal [13]).

A. Solution Space Variability

The solution space captures variability at the level of composition constructs of either component models or process languages. In particular, components models define variation points using parametric variability or enumerative variability. Approaches using parametric variability [14], [15], [16] manually define a fixed number of behaviour variants at the implementation-level during design-time. Hence, there is only one workflow with multiple branching structures. Furthermore, dynamic reconfiguration is needed to change the composition structure at runtime.

Only FX-MAN [17] enumerates all possible variants in the solution space at design-time. However, it does not support service composition, requires variation generators on top of compositions, and does not addresses variability of control flow (i.e., workflow variability) and workflow selection at runtime.

Approaches extending Process Modeling Languages allow the definition of control flow constructs (e.g., activities or gateways) as variation points whose variants are realized via model transformations [2]. Most of the approaches [18], [19], [20], [21] support control flow variability only at conceptual level as they operate on non-executable models. Only few approaches [22], [23] support control flow variability via executable models (e.g., YAWL or BPEL). The main drawback is that they operate on a single flat workflow which is customized by adding, removing or replacing business process fragments via reconfiguration rules. At runtime, workflows are customized using process flexibility (i.e., dynamic reconfiguration) [24].

Other approaches [25] extend business processes with support for parametric variability. However, they also require dynamic binding at runtime and the number of variants are limited as they are manually fixed at design-time.

B. Models@Runtime

Traditional Software Product Lines (SPL) [26] enable the modeling of families of related products (i.e., workflows). As variability is separated from the behavioural model, SPL requires linking a non-executable variability model with an executable software architecture. To do so, a developer needs to implement the product in such a way that the software

architecture matches the selected features. So, SPL naturally lacks mechanisms for changing behaviour at runtime.

Dynamic Software Product Lines (DSPL) [27] change behaviour at runtime whenever the context changes, by using models@runtime [28] to causally connect a variability model (typically a feature model [29] or an orthogonal variability model [30]) with a behavioural model (typically architectural units). To change behaviour, they bind variation points at runtime by selecting (i.e., activating or deactivating) features that best adapt to the current context. Thus, a set of features represents a behaviour variant, which is transformed into a software architecture using a transformation mechanism [29], [31]. Undoubtedly, such a mechanism increases the overhead for changing behaviour at runtime. Furthermore, DSPL requires dynamic reconfiguration of the running composition, as they also separate variability from behaviour.

Dynamic reconfiguration includes code substitution (e.g., parametrization or pre-processor directives) [32], [33], dynamic aspect weaving [34], [29], [35], [36], [1], [36], enabling/disabling services and connectors [37], [3], and component substitution [38], [39].

C. Discussion

Parametric variability is only suitable when all variants can be defined and implemented in advance. However, IoT systems require plenty of different alternative behaviours for adapting to the ever changing context, even though they operate under closed environments. For that reason, parametric variability is inconvenient for highly dynamic IoT environments.

Remarkably, DX-MAN does not require the manual definition of alternative behaviours since an infinite number of workflow variants simultaneously exist at the conceptual level of a composite service. As it is infeasible to implement and deploy infinite workflow variants, workflows are non-deployable and executable only. Exogenous connectors are the actual deployable entities (cf., [4]) which coordinate the execution of multiple workflow variants. Thus, our approach does not operate on a single flat workflow, but on a multi-level composite where there is a workflow space (with multiple workflows) at every level of the hierarchy.

Constraints are important to filter out the workflows that a designer considers invalid under a closed environment. Hence, DX-MAN supports the definition of constraints in a similar fashion to feature models, with the difference that constraints are directly applicable to system's behaviour. DX-MAN currently supports topological sorting (for sequencers) and logical constraints (for parallelizers). We do not explain them due to space constraints.

Models@runtime separate variability and behaviour to allow an independent reasoning of these concerns. However, as scale increases and dependencies become overwhelming, the relationship between features and architectural artefacts becomes unmanageable. Hence, models@runtime face several problems when coping with dependencies. Moreover, the separation between variability and behavior requires dynamic reconfiguration to maintain a causal relationship between both dimensions. Dynamic reconfiguration is undesirable for highly dynamic IoT environments, since it takes time to decide the actions to be done, performing those actions, ensuring state consistency, checking safeness and redeploying the running composition. Remarkably, DX-MAN does not require any means to connect variability with behaviour as those dimensions are mixed in the definition of composite services, thereby avoiding the need of dynamic reconfiguration.

We previously presented a preliminary version of DX-MAN (cf. [5]). In this paper we described new semantics for supporting variability using workflow spaces. We also presented detailed examples to explain autonomicity, and a new composition connector called *exclusive selector*. Furthermore, we extended DX-MAN with capabilities for changing behaviour at runtime using MAPE-K loops.

A MAPE-K loop controls the behaviour of a composite service and is defined according to the expected goal of the managed composite. We particularly focus on the executor component which do not perform dynamic reconfiguration, but only execute a concrete workflow tree (i.e., a workflow variant) for adapting to different contexts.

Although our examples show autonomicity only in the context of IoT, DX-MAN can be used for other domains such as robotics, unmanned space or e-commerce. It is important to mention that we emphasize on the semantics of our model, rather than focusing on a particular implementation. Nevertheless, an implementation of DX-MAN is available at https://gitlab.cs.man.ac.uk/mbaxrda2/dxman.

VII. Conclusions and Future Work

This paper extended the semantics of the DX-MAN model by mixing variability with behaviour in composite services. In particular, composition connectors are variability operators that define composite workflow spaces containing an infinite number of workflow variants which represent alternative composite service behaviours. Thus, composite services define an infinite number of Turing machines at once in the design phase.

A MAPE-K manages a composite service behaviour and selects the workflow variant that best adapts to the current context. As workflows are non-deployable and executable only, the executor changes a composite service behaviour by executing the selected variant instead of dynamically reconfiguring the whole workflow. The variant is a concrete workflow tree built at runtime from an abstract workflow tree (defined at design-time). Composition connectors are the actual deployable entities which coordinate the execution of multiple workflows, thereby reusing the same deployment configuration for multiple executions.

We demonstrated the autonomic capabilities of DX-MAN using a case study in the domain of smart homes. Our results indicate that DX-MAN is a promising model for autonomic IoT systems. Nevertheless, there are some open issues.

DX-MAN currently enables control flow variability, making it suitable for actuating operations that do not require any data, e.g., switching the lights on. We plan to investigate novel

ways of incorporating data flow variability by leveraging the separation of autonomicity, control, data and computation.

DX-MAN is suitable for closed environments only where the designer understands the context in which the system is deployed. We are currently investigating novel ways to dynamically evolve a DX-MAN composition, so as to enable the emergence of new workflow spaces at runtime. Evolution is indeed another important characteristic of autonomic IoT systems, in addition to workflow variability.

REFERENCES

[1] G. H. Alférez and V. Pelechano, "Achieving autonomic Web service compositions with models at runtime," *Computers & Electrical Engineering*, vol. 63, pp. 332–352, Oct. 2017.

[2] M. L. Rosa *et al.*, "Business Process Variability Modeling: A Survey," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 2:1–2:45, Mar. 2017.

[3] H. Gomaa and M. Hussein, "Dynamic Software Reconfiguration in Software Product Families," in *Software Product-Family Engineering*, ser. Lecture Notes in Computer Science, F. J. van der Linden, Ed. Springer Berlin Heidelberg, 2004, pp. 435–444.

[4] D. Arellanes and K.-K. Lau, "Exogenous Connectors for Hierarchical Service Composition," in *IEEE SOCA*, 2017, pp. 125–132.

[5] D. Arellanes and K.-K. Lau, "Algebraic Service Composition for User-Centric IoT Applications," in *ICIOT 2018*, ser. Lect. Notes Comp. Sci. Springer Int. Pub., 2018, pp. 56–69.

[6] D. Arellanes and K.-K. Lau, "D-XMAN: A Platform For Total Compositionality in Service-Oriented Architectures," in *IEEE SC2*, 2017, pp. 283–286.

[7] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[8] J. Hess *et al.*, "Poisson-driven dirt maps for efficient robot cleaning," in *2013 IEEE International Conference on Robotics and Automation*, May 2013, pp. 2245–2250.

[9] M. A. Yakoubi and M. T. Laskri, "The path planning of cleaner robot for coverage region using Genetic Algorithms," *Journal of Innovation in Digital Ecosystems*, vol. 3, no. 1, pp. 37–43, Jun. 2016.

[10] B. Dugarjav *et al.*, "Scan matching online cell decomposition for coverage path planning in an unknown environment," *Int. J. Precis. Eng. Manuf.*, vol. 14, no. 9, pp. 1551–1558, Sep. 2013.

[11] K. Kakousis *et al.*, "Optimizing the Utility Function-Based Self-adaptive Behavior of Context-Aware Systems Using User Feedback," in *On the Move to Meaningful Internet Systems: OTM 2008*, ser. Lecture Notes in Computer Science, R. Meersman and Z. Tari, Eds. Springer Berlin Heidelberg, 2008, pp. 657–674.

[12] R. R. Filho and B. Porter, "Defining Emergent Software Using Continuous Self-Assembly, Perception, and Learning," *ACM Trans. Auton. Adapt. Syst.*, vol. 12, no. 3, pp. 16:1–16:25, Sep. 2017.

[13] E. Bruneton *et al.*, "The FRACTAL component model and its support in Java," *Software: Practice and Experience*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[14] A. Haber *et al.*, "Hierarchical Variability Modeling for Software Architectures," in *2011 15th International Software Product Line Conference*, Aug. 2011, pp. 150–159.

[15] R. v. Ommering *et al.*, "The Koala component model for consumer electronics software," *Computer*, vol. 33, no. 3, pp. 78–85, Mar. 2000.

[16] E. M. Dashofy *et al.*, "A Comprehensive Approach for the Development of Modular Software Architecture Description Languages," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 2, pp. 199–245, Apr. 2005.

[17] C. Qian and K. Lau, "Enumerative Variability in Software Product Families," in *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, Dec. 2017, pp. 957–962.

[18] M. La Rosa *et al.*, "Configurable multi-perspective business process models," *Information Systems*, vol. 36, no. 2, pp. 313–340, Apr. 2011.

[19] I. Reinhartz-Berger *et al.*, "Extending the Adaptability of Reference Models," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 40, no. 5, pp. 1045–1056, Sep. 2010.

[20] A. Hallerbach *et al.*, "Capturing Variability in Business Process Models: The Provop Approach," *J. Softw. Maint. Evol.*, vol. 22, no. 6-7, pp. 519–546, Oct. 2010.

[21] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," in *Generative Programming and Component Engineering*, ser. Lecture Notes in Computer Science, R. Glück and M. Lowry, Eds. Springer Berlin Heidelberg, 2005, pp. 422–437.

[22] F. Gottschalk *et al.*, "Configurable workflow models," *Int. J. Coop. Info. Syst.*, vol. 17, no. 02, pp. 177–221, Jun. 2008.

[23] A. Kumar and W. Yao, "Design and management of flexible process variants using templates and rules," *Computers in Industry*, vol. 63, no. 2, pp. 112–130, Feb. 2012.

[24] R. Cognini *et al.*, "Business process flexibility - a systematic literature review with a software systems perspective," *Inf Syst Front*, vol. 20, no. 2, pp. 343–371, Apr. 2018.

[25] M. Koning *et al.*, "VxBPEL: Supporting variability for Web services in BPEL," *Information and Software Technology*, vol. 51, no. 2, pp. 258–269, Feb. 2009.

[26] K. C. Kang and a. P. Donohoe, "Feature-oriented product line engineering," *IEEE Software*, vol. 19, no. 4, pp. 58–65, Jul. 2002.

[27] S. Hallsteinsen *et al.*, "Dynamic Software Product Lines," *Computer*, vol. 41, no. 4, pp. 93–95, Apr. 2008.

[28] G. Blair *et al.*, "Models@ run.time," *Computer*, vol. 42, no. 10, pp. 22–27, Oct. 2009.

[29] B. Morin *et al.*, "Models@ Run.time to Support Dynamic Adaptation," *Computer*, vol. 42, no. 10, pp. 44–51, Oct. 2009.

[30] N. Bencomo *et al.*, "Genie: Supporting the Model Driven Development of Reflective, Component-based Adaptive Systems," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 811–814, event-place: Leipzig, Germany.

[31] I. Schaefer *et al.*, "Delta-Oriented Programming of Software Product Lines," in *Software Product Lines: Going Beyond*, ser. Lecture Notes in Computer Science, J. Bosch and J. Lee, Eds. Springer Berlin Heidelberg, 2010, pp. 77–91.

[32] B. Morin *et al.*, "Taming Dynamically Adaptive Systems using models and aspects," in *2009 IEEE 31st International Conference on Software Engineering*, May 2009, pp. 122–132.

[33] C. Parra *et al.*, "Context Awareness for Dynamic Service-oriented Product Lines," in *Proceedings of the 13th International Software Product Line Conference*, ser. SPLC '09. Pittsburgh, PA, USA: Carnegie Mellon University, 2009, pp. 131–140, event-place: San Francisco, California, USA.

[34] G. H. Alférez *et al.*, "Dynamic adaptation of service compositions with variability models," *Journal of Systems and Software*, vol. 91, pp. 24–47, May 2014.

[35] L. Baresi *et al.*, "Service-Oriented Dynamic Software Product Lines," *Computer*, vol. 45, no. 10, pp. 42–48, Oct. 2012.

[36] F. Fleurey *et al.*, "A Generic Approach for Automatic Model Composition," in *Models in Software Engineering*, ser. Lecture Notes in Computer Science, H. Giese, Ed. Springer Berlin Heidelberg, 2008, pp. 7–15.

[37] C. Cetina *et al.*, "Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes," *Computer*, vol. 42, no. 10, pp. 37–43, Oct. 2009.

[38] J. Floch *et al.*, "Using architecture models for runtime adaptability," *IEEE Software*, vol. 23, no. 2, pp. 62–70, Mar. 2006.

[39] J. White *et al.*, "Creating self-healing service compositions with feature models and microrebooting," *International Journal of Business Process Integration and Management*, vol. 4, no. 1, p. 35, 2009.