Managing Run-Time Variability in Robotics Software by Modeling Functional and Non-functional Behavior

Alex Lotz[1], Juan F. Inglés-Romero[2], Cristina Vicente-Chicote[3], and Christian Schlegel[1]

[1] University of Applied Sciences Ulm, Germany
{lotz,schlegel }@hs-ulm.de
[2] Universidad Politécnica de Cartagena, Spain
juanfran.ingles@upct.es
[3] QSEG, Universidad de Extremadura, Spain
cristinav@unex.es

Abstract. Service robots act in open-ended and natural environments. Therefore, due to the huge number of potential situations and contingencies, it is necessary to provide a mechanism to express dynamic variability at design-time that can be efficiently resolved on the robot at run-time based on the then available information. In this paper, we present a modeling process to separately specify at design-time two different kinds of dynamic variability: (i) variability related to the robot operation, and (ii) variability associated with QoS. The former provides robustness to contingencies, maintaining a high success rate in robot task fulfillment. The latter focuses on the quality of the robot execution (defined in terms of non-functional properties like safety or task efficiency) under changing situations and limited resources. We also discuss different alternatives for the run-time integration of the two variability management mechanisms, and show real-world robotic examples to illustrate them.

Keywords: Variability Management, Modeling Run-Time Variability, Service Robotics, SmartTCL, VML.

1 Introduction

Service robots (e.g. companion, elder care, home health care, or co-worker robots) are expected to robustly and efficiently fulfill different tasks in complex environments (such as domestic, outdoor or crowded public spaces). Real-world environments are inherently open-ended and show a huge number of variants and contingencies. Thus, service robots need to know how to flexibly plan their sequence of actions in order to fulfill their tasks, taking into account changes in the environment, noisy perception and execution failures. For instance, a robot navigating in a building needs to reactively avoid dynamic obstacles in its local surrounding. Besides, it might need to reconsider its plan of how to get to its destination, e.g., in case the planned route is blocked. The management of this *variability in operation* provides the robot with a high degree of robustness and allows it maintaining a high success rate in task fulfillment. On the other hand, there is typically a wide range of possibilities to succeed in the same task. Among these possibilities, some might be better than others according to quality criteria defined by the designer (e.g., in terms of resource consumption, safety, performance, etc.). For example, since robots are equipped with limited resources, they need to know how to spend them in the most appropriate way. Thus, if a robot is running out of battery, it might be a good idea to prioritize power consumption over task efficiency. The management of this *variability in quality* improves the overall robot execution performance.

Addressing variability in robotic software is nothing new. However, it has been traditionally managed in an ad-hoc way, i.e., developers try to predict future execution conditions and implement specific mechanisms to deal with each particular situation, spreading the variability management rationale thoughout their application code. Among other issues, this usually leads to increased complexity, poor reuse, and it hinders the extensibility and maintenance of the applications. This motivates a different approach [1]: (i) we need to make it as simple as possible for the designer to express variability at design-time, and (ii) we need the robot to be able to bind variability at run-time, based on the then available information. At design-time, we propose to use two different Domain Specific Languages (DSLs), each one for modeling one of the previously described varibility kinds: SmartTCL [2] for *variability in operation*, and VML [1] for *variability in quality*. This way we encourage the separation of the two concerns: one for modeling *how* to coordinate the actions (e.g., a flexible plan considering contingencies), and another one for modeling *what is a good way* of achieving a task (e.g., in terms of non-functional properties like safety or power consumption). At run-time, we separate the variability management in two orthogonal mechanisms: (i) sequencing the robot's actions to handle *variability in operation*, and (ii) optimizing the non-functional properties for *variability in quality*. These two mechanisms enable the robot to decide on proper behavior variations by applying the design-time model information and taking the current situation into account. This approach improves the robustness and the task execution quality, optimizes robot performance and cleverly arranges complexity and efforts between design-time and run-time.

In this paper, we present a modeling process to separately specify *variability in operation* using SmartTCL and *variability in quality* using VML. Besides, as one of the core contributions of this paper compared to our previous work on this topic [1], we analyze three different alternatives for a run-time integration of the two proposed variability management mechanisms in a safe and consistent way. We will describe the lessons learned and discuss the benefits and drawbacks of each alternative. In order to underpin the feasibility and the benefits of the proposal we provide a real-world case study in a robotics scenario. Despite the fact that our proposal is inspired by the robotic domain, we believe that the basic ideas we present in this paper are interesting and general enough to be considered in other domains.

2 Real-World Robotics Scenario

In order to demonstrate the variability modeling capabilities of the two individual languages, SmartTCL and VML, we use the *Butler*[1] scenario. In this scenario we have a robot serving people. It carries out typical butler activities like taking orders, fetching beverages, cleaning up after customers left the place, etc. Although we have repetitive tasks in the scenario, the overall sequence of actions is never the same. Instead, the service robot must adjust its behavior according to changes in the environment, human orders, etc.

To achieve this flexibility in the robot operation, SmartTCL allows designers to model possible contingencies of the scenario. For instance, in the *clean up the table* activity, the robot has to take objects like glasses and cups to the kitchen, and place others, like tetra packs or empty cans, in a trash bin. Then, what happens if the robot finds unexpected objects on the table? Although the robot can presume which objects are left on a table from the previous orders, the real situation might be quite different, as customers might have left their glasses on different tables or forgotten personal objects such as a mobile phone. Besides, typically, the sequence of appropriate actions for the clean up task strongly depends on each particular situation, and needs to take into account both robot limitations and constraints (e.g., its physical capacity to carry only a limited weight or the maximum number of cups it can stack into each other to safely carry them to the kitchen), and application-specific information (e.g., which objects must be thrown away and which ones must be cleaned and reused). Thus, the most robust way for a service robot to operate is to react on each situation with as little assumptions as possible. This motivates an approach where the behavior of the robot must be as flexible as possible, just defining the strategies for how to react on situations by finding appropriate sequences of actions to achieve, e.g., the clean up task.

Now, imagine that we want to optimize the *coffee delivery*[2] service. The robot has to trade-off various aspects to come up with an improved quality of service. Thus, it needs to be able to select an appropriate velocity to properly fulfill its task according to further issues like safety or energy consumption: (i) customers are satisfied only if the coffee has at least a certain temperature, but prefer it as hot as possible; thus, serving fast is relevant, (ii) however, the maximum allowed velocity is bound due to safety issues (hot coffee) and also by battery level, (iii) since coffee cools down depending on the time travelled, a minimum required average velocity (depending on distance to customer) is needed, although driving slowly makes sense in order to save energy, (iv) nevertheless, fast delivery can increase the volume of sales. Although the main functionality of the robot is to deliver coffee, regardless of how well it is performed, VML can help designers to model, on top of this functionality, QoS policies based on (often conflicting) non-functional properties. As a result, the variability (e.g., robot maximum allowed

[1] Butler scenario video: `http://www.youtube.com/user/RoboticsAtHsUlm`
[2] Coffee scenario video: `http://www.youtube.com/watch?v=-nmliXl9kik`

velocity) is bound at run-time to optimize these policies according to the current application context (e.g., the battery level or how crowded the coffee shop is).

Inspired on the above example, we consider that the Butler scenario takes place in a room with two coffee machines located in different positions. When someone asks the robot for a cup of coffee it must decide: (i) which coffee machine to use, and (ii) its maximum allowed velocity. This decision is made at run-time in order to improve the quality of the service taking into account power consumption (e.g., when the battery is low the system must optimize power consumption using the nearest coffee machine) and performance (e.g., trying to get the highest value for maximum allowed velocity in order to reach the goal faster). Obviously, maximizing performance while simultaneously minimizing power consumption imposes conflicting requirements. To deal with this, VML allows expressing, at design-time, the existing dependencies among conflicting requirements such that, at run-time, the robot can find the right balance among them.

3 Modeling Variability with SmartTCL

We use the Task Coordination Language (SmartTCL [2]) to model *variability in operation*. The main purpose of SmartTCL is to define rules and strategies that specify *how* the system behaves when accomplishing different tasks. SmartTCL allows to react to changes in the environment and to adjust task execution according to the current situation.

3.1 SmartTCL Syntax

The SmartTCL EBNF grammar is defined in Listing 1.1. The main element of SmartTCL is the *Task Coordination Block* (TCB). A TCB represents an abstract task (e.g., moving to a generic location), and its function is to *orchestrate* (configure and activate) the system components to execute the proper primitive actions needed to achieve this task. A TCB is defined by its signature, consisting of the name and the in/out parameters and, optionally, a *precondition* clause and a *priority* that, when available, help selecting the most appropriate TCB at run-time. The *body* of a TCB contains, at least, an *action* block, a *plan* or both. An *action* block is used to define primitive behaviors encoded in Lisp. A *plan* is used to establish a parent/child relationship between TCBs and, thus, to create complex behaviors. This enables SmartTCL to define recursion (with the termination clause encoded as the precondition) and to create *task-trees* consisting of TCBs as nodes. The plan is not static, but can be dynamically adjusted at run-time, e.g., by asking a symbolic planner (which allows to generate action plans at run-time) for a sequence of TCBs. The default execution semantics for the TCBs in a *plan* is to execute them one after the other in sequence. Alternatively, it is possible to execute all the TCBs in a *plan* in *parallel* (waiting until they have all finished), or use the *one-of* semantics (again in parallel but, this time, exiting as soon as one of the TCBs finishes). In addition, the *body* of a

```
SmartTCL ::= ( TCB | [EventHandler] | [RuleDef] )+
(* TCB definition *)
TCB ::= '(' Signature Body ')'
TCB_NAME ::= ID
LISP_CODE ::= STRING
(* TCB-Signature definition *)
Signature ::= 'define-tcb' '(' TCB_NAME InParams '=>' OutParams ')'
     [Precondition] [Priority]
InParams ::= ('?' ID)*
OutParams ::= ('?' ID)*
Precondition ::= '(' 'precondition' '(' LISP_CODE ')' ')'
Priority ::= '(' 'priority' INT ')'
(* TCB-Body definition *)
Body ::= [Rules] (ActionBlock [Plan] | [ActionBlock] Plan) [AbbortActions]
Rules ::= '(' 'rules' '(' (ID)+ ')' ')'
ActionBlock ::= '(' 'action' '(' LISP_CODE ')' ')'
AbortActions ::= '(' 'abort-action' '(' LISP_CODE ')' ')'
Plan ::= '(' 'plan' '(' ['parallel'|'one-of'] (TCB_NAME)+ ')' ')'
(* EventHandler definition *)
EventHandler ::= '(' 'define-event-handler' '(' ID ')' ActionBlock ')'
(* Rule block definition *)
RuleDef ::= '(' 'define-rule' '(' TcbBinding TcbEvent ActionBlock ')' ')'
TcbBinding ::= '(' 'tcb' '(' TCB_NAME InParams '=>' OutParams ')' ')'
TcbEvent ::= '(' 'tcb-return-value' '('TcbEventCode'('TcbEventDescription')'')'')'
TcbEventCode ::= ('SUCCESS' | 'ERROR')
TcbEventDescription ::= (STRING)*
```

Listing 1.1. EBNF grammar for SmartTCL

TCB can optionally contain a sequence of *rules* (see below) or a block of *abort-actions*. The latter implement cleanup procedures in case the TCB is aborted before completion.

In addition to TCBs, SmartTCL defines *EventHandlers* and *Rules*. *EventHandlers* are used to receive feedback from the components in the system. This feedback can signal, e.g., successful completion or a failure in the execution of a previously triggered action. In any case, the *EventHandler* executes a block of actions to appropriately react on that event. A *rule* is a very handy mechanism to react on contingencies during the execution of the TCBs in a plan. This mechanism is comparable with C++ Exceptions. If one of the child TCBs has a problem during its execution that cannot be solved locally, it can propagate an error message up the hierarchy, which is then caught by the first fitting *rule*. For each TCB, various rules can be defined (indicated by the binding part of the rule and associated to a certain event) to provide different strategies to react on contingencies. A parent TCB activates a set of rules for its child TCBs in order to create appropriate recovery strategies. This considerably improves the reuse and flexibility of the TCBs in different scenarios.

3.2 SmartTCL Execution Semantics

At run-time, on the robot, SmartTCL plays the coordinator role by orchestrating software components. Thereby, each TCB represents a certain, consistent system state with all the configuration parameters for individual components.

The main functionality of SmartTCL is illustrated by the *clean-up table* scenario introduced in Sec. 2. The whole scenario is modelled on the robot using

SmartTCL. There, a user operating with the robot can command it to clean up the dinner table, which activates the TCB labelled in Listing 1.2 and in Fig. 1 as (1)). This TCB has an input parameter named `?location`. From the parent TCB this parameter is set to `dinner-table`, which is a placeholder that can be resolved in the knowledge base (like in (3)). It is worth noting that, in our case, a knowledge base is simply a databse (a memory) to store key/value pairs (for e.g. the robot model, the TCBs, environment model, etc.).

```
(define-tcb (cleanup-table ?location)
  (rules (rule-action-cleanup-table-failed rule-action-cleanup-table-empty
    rule-action-cleanup-say-success ))
  (plan (
    (tcb-approach-location ?location)
    (tcb-say "I have reached the table, and will look for objects to clean up.")
    (tcb-cleanup-table))))
```

Listing 1.2. Cleanup table TCB definition

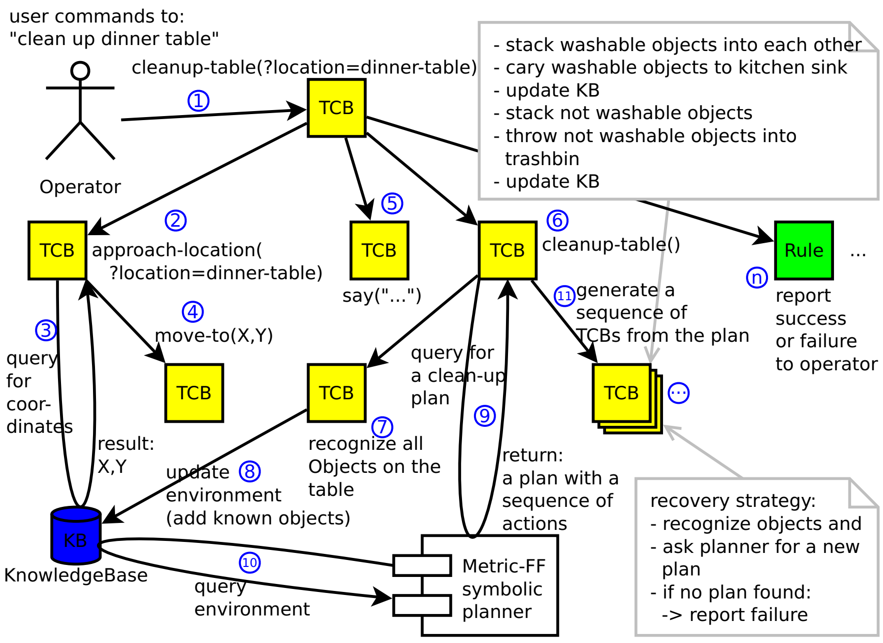
user commands to:
"clean up dinner table"
cleanup-table(?location=dinner-table)
Operator
1
TCB
- stack washable objects into each other
- cary washable objects to kitchen sink
- update KB
- stack not washable objects
- throw not washable objects into trashbin
- update KB
2
approach-location(?location=dinner-table)
5
TCB
say("...")
6
cleanup-table()
Rule
...
n
report success or failure to operator
3
query for coor-dinates
result: X,Y
4
move-to(X,Y)
query for a clean-up plan
11
generate a sequence of TCBs from the plan
...
7
recognize all Objects on the table
9
8
update environment (add known objects)
return: a plan with a sequence of actions
KB
KnowledgeBase
10
query environment
Metric-FF symbolic planner
recovery strategy:
- recognize objects and
- ask planner for a new plan
- if no plan found:
-> report failure

Fig. 1. Snapshot of a task-tree at run-time from the clean-up scenario example

As shown in Listing 1.2, the *cleanup-table* TCB has no *action* block, but it defines a *plan* comprising other TCBs to approach a location (2) (in this case the dinner-table), to announce via speech that the robot is now going to look for objects to clean up (5), and then to execute the TCB `cleanup-table` (6), which is different from its parent because of its different signature. In addition, a list of *rules* is activated which, in this case, altogether belongs to the child

TCB `cleanup-table`. These rules define the behavior of this TCB according to certain results in the execution. For instance, if something goes wrong while cleaning up the table, the first rule can trigger a re-planning or can delete the current plan and report the failure to the operator. In case the cleaning up succeeds, the robot can update its internal environment representation and report the success to the operator (see also (n) in Fig. 1).

It is important to notice that task-trees are created and evolve (according to changes in the environment, execution failures, etc.) only at run-time. It is also noticeable that, in contrast to regular finite state machines, we do not model the transitions between the TCBs. Instead we define rules and strategies that specify how a TCB can be expanded and refined at run-time taking the then available information into account. This leads to a much more flexible and robust system behavior. It is even possible and a regular case to generate complete (sub)trees by asking a symbolic planner (see (9) and (11)). In this example, the planner returns a sequence like: stack cup A into cup C and then cup C into cup B, etc. This is a powerful mechanism we call *expert usage* that will be handy for interacting with VML (see Sec. 5). In summary, we use *experts*, *rules* and *event handlers* to design *variability in operation*.

4 Modeling Variability with VML

In this section we introduce the *Variability Modeling Language* (VML) that provides a mechanism to express *variability in quality*, that is, how a system should adapt at run-time to maintain or improve the system execution quality under changing conditions. The current version of VML has been developed using a *Model-Driven Engineering* (MDE) approach. We have created a textual editor for VML using the Xtext framework[3], including some advanced features such as syntax checking, coloring and a completion assistant.

In a VML model, we first need to define the variation points. Aligned with *Dynamic Software Product Lines* (DSPL) [3], VML variation points represent points in the software where different variants might be chosen to derive the system configuration at run-time. Therefore, variation points determine the decision space of VML, i.e., the answer to *what* can change. As shown in Listing 1.3, variation points (`varpoint`), as all the other VML variables, belong to a certain data type. VML includes three basic data types: enumerators, ranges of numbers and booleans. For instance, `maximumVelocity` is a variation point to limit the maximum velocity of the robot, which takes values from 100 to 600 with precision 10 mm/s. Similarly, the `coffeeMachine` variation point is an enumerator that gathers the two available coffee machines that can be used by the robot: `COFFEE_MACHINE_A` and `COFFEE_MACHINE_B`. Once we have identified the variation points, context variables (`context`) are used to express situations in which variation points need to be adapted. Listing 1.3 shows five context variables: (i) the battery level (integer value in the range 5-100), (ii) distance to

[3] Xtext: `www.eclipse.org/Xtext`

each coffee machine (real number in the range 0-20 with precision 0.1), and (iii) waiting time at each coffee machine (integer in the range 10-300), taking into account the operation time of the machine (constant time) and the time the robot has to wait because there are other users (robots or people) waiting for it (variable time).

```
/* Contexts variables */
context ctx_battery                : number [0:1:100];
context ctx_distanceMachine_A      : number [0:0.1:20];
context ctx_distanceMachine_B      : number [0:0.1:20];
context ctx_waitingTimeMachine_A   : number [10:1:300];
context ctx_waitingTimeMachine_B   : number [10:1:300];

/* Auxiliary variables */
var timeMachine_A := ctx_waitingTimeMachine_A + ctx_distanceMachine_A/600;
var timeMachine_B := ctx_waitingTimeMachine_B + ctx_distanceMachine_B/600;

/* ECA rules */
rule lowBattery_NearMachineA :
   ctx_battery < 15 and ctx_distanceMachine_A < ctx_distanceMachine_B
   => coffeeMachine = @coffeeMachine.COFFEE_MACHINE_A;
rule lowBattery_NearMachineB :
   ctx_battery < 15 and ctx_distanceMachine_A >= ctx_distanceMachine_B
   => coffeeMachine = @coffeeMachine.COFFEE_MACHINE_B;
rule high_EFF_coffeeMachA :
   ctx_battery >= 15 and timeMachine_A  > timeMachine_B
   => coffeeMachine = @coffeeMachine.COFFEE_MACHINE_A;
rule high_EFF_coffeeMachB :
   ctx_battery >= 15 and timeMachine_A <= timeMachine_B
   => coffeeMachine = @coffeeMachine.COFFEE_MACHINE_B;

/* Properties to optimize */
property performance : number[0:1:100] maximized {
  priorities:
    f(ctx_battery) = max(exp(-1*ctx_battery/15), 10 sec) - exp(-1*ctx_battery/15);
  definitions:
    f(maximumVelocity) = maximumVelocity;
}
property powerConsumption : number[0:1:100] minimized {
  priorities:
    f(ctx_battery) = exp(-1 * ctx_battery/15);
  definitions:
    f(maximumVelocity) = exp(maximumVelocity/150);
}

/* Variation points */
varpoint maximumVelocity : number [100:10:600];
varpoint coffeeMachine   : enum { COFFEE_MACHINE_A, COFFEE_MACHINE_B };
```

Listing 1.3. VML Model for choosing Coffee Machine

At this point, we need to define how variation points are set according to the context. This is achieved through properties (`property`) and Event-Condition-Action (ECA) rules (`rule`). Properties specify the features of the system that need to be optimized, i.e., minimized or maximized. Properties are defined using two functions: *priorities* and *definitions*. Definitions characterize the property in terms of variation points (i.e., definitions are the objective functions to be optimized). For instance, in Listing 1.3, we define the performance property as a linear function of the maximum velocity variation point (the faster the robot accomplishes its task, the better its performance). Similarly, the power consumption property also depends (in this case, exponentially) on the maximum velocity (the faster the robot moves, the higher its power consumption). It is worth noting that property *definitions* are characterized using the technical specifications of the hardware (e.g., to know how much power each component consumes), simulation or empirical data from experiments. On the other hand, *priorities* describe the importance of each property in terms of one or more context variables (i.e., priorities weight the objective functions). For instance, power consumption becomes more and more relevant as the robot battery decreases. In fact, when the battery is full, power consumption is not considered an issue and, as a consequence, the priority of this property in that context is very small. Opposite to definitions, priorities are characterized in a more subjective way, depending on the designer experience.

Regarding the ECA rules, they define direct relationships between context variables and variation points. As shown in Listing 1.3, the left-hand side of a rule expresses a trigger condition (depending on one or more context variables) and its right-hand side sets the variation point. For example, the decision of which coffee machine to select in each situation is modeled using rules. Basically, the first two rules are applied when the battery is low (less than 15%) to select the nearest coffee machine (reducing travel distance lowers power consumption when the battery is critical). Conversely, if the battery is high enough, the last two rules select the machine with lower waiting time (reducing the coffee delivery time improves performance when the battery is not an issue).

Regarding execution semantics, VML models specify a constrained optimization problem, that is, it describes the global weight function that optimizes the variation points to improve the overall system quality. This global function is obtained by aggegating the property definitions (terms to be optimized), weighted by their corresponding priorities. Besides, the ECA rules state constraints that need to be satisfied. Therefore, in order to execute a VML model, we transform it into *MiniZinc* [4] (a constraint programming language), so that the generated code is used as an input by the *G12 Constraint Programming Platform*[4] to solve the constraint problem. At this point, it is worth noting that VML variation points and contexts are high-level variables that somehow abstract architectural elements (e.g., components or component parameters). For instance, the maximum velocity is linked to a parameter of the motor component, and the battery level is obtained from a sensor component. This abstraction allows VML to be independent of the underlying architecture, what, among other benefits, enables the reuse of the specification. However, when it comes the time to map the abstract VML elements to the actual system architecture, we must carefully take into account how this might interact with the decisions made by SmartTCL. Next section explains how both variability management mechanisms have been finally integrated in a safe and consistent way, after assessing different approaches.

[4] G12: `www.g12.csse.unimelb.edu.au`

5 Run-Time Integration of SmartTCL and VML

To this point, we have explained how to use SmartTCL and VML to separately specify the *variability in operation* and the *variability in quality*, respectively. This section addresses one of the key issues when using both variability management mechanisms together, i.e., how to integrate them in a safe and consitent way. Next, we describe chronologically the integration approaches we followed to deal with this issue.

As detailed in Sec. 3, the role of SmartTCL is to orchestrate the changes in the software architecture, i.e., to configure, activate and deactive the components, according to an action plan, to enable the robot to fulfill its tasks. On the other hand, the VML engine sets a number of variation points (with impact in the architecture components or in some of their parameters), so that the overall QoS delivered by the robot is optimized. Our first integration approach was to enable both mechanisms to have impact on the software architecture. However, although we can advise the designer to create SmartTCL and VML models having orthogonal decision spaces, formally checking this orthogonality at design-time is a hard problem. We evaluated the possibility of implementing a tool that could help the designer in the creation of mutually consistent SmartTCL and VML models (i.e., guaranteeing that both models would not produce conflicting decisions at runtime). However, as the number of potential situations the system could have to face is unbounded and first known at run-time, the potential decisions of both mechanisms in response to those situations are also unbounded. Thus, we discarded this initial approach as it was not feasible in practice.

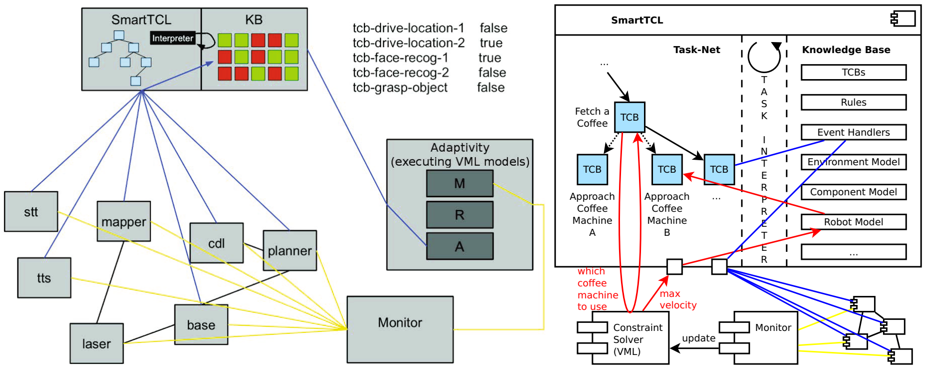

Fig. 2. On the left: discrete TCBs to handle variants during optimization. On the right: expert usage and continuous updates

In the robotics domain, one general approach to face decision conflicts is to define a single master in the system, which is responsible for making the final decisions at run-time. In our case, there is a natural hierarchy already available. Thereby, SmartTCL models all functional aspects related to the *variability in operation*. In designing such models, typically one finds several solutions with similar success expectations regarding the goals of the tasks. From now on, we will call these different solutions for the same task *variants*. These variants can be used as an input for the VML engine which, at run-time, will select one out of them (the best possible one) by optimizing the non-functional properties defined in the VML model. This way, the VML engine will advise SmartTCL on its decisions but will not interfer with the overall system functionality. As shown in Fig. 2 (on the left) we have implemented such *variants* as extra TCBs, which have an additional boolean flag indicating whether the TCB is activated (green boxes) or not (red boxes). At run-time, the VML engine (executed by the Adaptivity component): (i) monitors (M) the system to determine the current situation, (ii) then reasons (R) about the appropriate adjustments, and (iii) activates (A) one out of the alternative TCBs. SmartTCL makes the final decision using only the TCBs that are active in that moment. This already allows to handle problems similar to the coffee delivery example, where one TCB would be for approaching coffee machine A and another TCB for coffee machine B. Although this approach works fine for simple scenarios, where discrete decisions must be taken, it is highly inefficient for continuous variation points. For example, the decision on the maximum allowed velocity would require: (i) deciding about the most appropriate discretization for the continuous variable, and (ii) creating a TCB for each discrete value. One could define a few discrete values like SLOW, MEDIUM and FAST, but the problem would remain unsolved when trying to assign concrete values to these labels in all potential situations. Another limitation of this approach is that all the variability must be statically defined at design-time (as concrete TCBs) and cannot be adjusted anymore at run-time according to different situations.

The limitations of the previous method motivated the need for a different approach that could directly deal with numeric and even floating point values. This approach follows the *subsidiarity principle*. Similar to a company where budgets are assigned down the hierarchy, and the lower departments manage their limited budgets without conflicting with the overall company decisions, a similar approach is applied in our software architecture. Therefore, we define contexts and variation points as the two interfaces between SmartTCL and VML. A *context* represents the current situation and the robot state. Contexts can be determined either directly from the knowledge base or deduced from an additional (monitoring) component, which infers certain information out of the raw data available in the system. A *variation point* is then a system variable (allowing either discrete or continuous values), whose boundaries are not static, but can be adjusted at run-time from SmartTCL according to the current functional needs. The remaining variability is then used by VML to decide on concrete values (subsidiarity principle). At run-time there are two mechanisms that implement this behavior. As mentioned in Sec. 3, SmartTCL allows to use different *experts* during execution (see Fig. 2 on the right). In our case, one of these experts is going to be the VML engine (constraint solver). For example, before expanding a TCB for fetching a coffee, SmartTCL sends a query to the VML engine expert asking for advise on which is the best coffee machine to go. SmartTCL will attach the pertinent context information in the query so that the VML engine can make an informed decision. For the continuous variation points a different mechanism is used. For example, for the selection of the maximum allowed velocity, an additional variable in the knowledge base is used (the robot model in Fig. 2 on the right). Thereby, SmartTCL adjusts this variable according to its functional needs, and VML is only allowed to set this variable within the remaining decision space. The VML engine is triggered whenever the monitor detects a situation that requires to adjust the maximum allowed velocity (e.g., when the environment is crowded). When this happens, the constraint solver of the VML engine calculares the optimum value for the variation point, and informs SmartTCL which, in turn, updates the knowledge base. In summary, this approach enables SmartTCL to be aware of VML models, and to ensure consistency (i) either by asking for advice about the possible alternatives, not conflicting with the overall task, or (ii) by propagating an already constrained variable to VML so that it can further constrain it without conflicting with the operational part.

6 Related Work

Service robotics is a challenging domain that exhibits a clear need for modeling and managing variability: robots should be able to change their behavior in response to changes in their operation or their environment. One step to solve this problem is to introduce mechanisms to model robot tasks independently of the components. In this sense, some work [5, 6] aims to rapidly compose complex robot behaviors based on state machines. These approaches support static composition of behaviors with little capabilities for situation dependent task adjustments. Therefore, designers need to include in the description of the functionality, which contingencies may occur for each situation, with almost no reuse. Conversely, SmartTCL allows to easily model dynamic task trees, which are rearranged at run-time according to the current situation.

Some other works have been applied to robotics, although they are not focused on this domain. Among them, DiaSpec [7] is a DSL to specify Sense/ Compute/ Control (SCC) applications, where the interaction with the physical environment is essential. DiaSpec allows designing applications independently of the underlying architecture, describing entities (e.g., components) and controllers, which execute actions on entities when a context situation is reached. The DiaSpec compiler generates Java code that needs to be completed with the application logic. Like SmartTCL and VML, DiaSpec platform-independence enables specification reuse. However, DiaSpec adaptation mechanisms completly rely on ECA-based rules and do not support any kind of optimization. Other works, like [8] and [9], have introduced the term architecture-based adaptation, applied to component-based robotic systems. In these works, components are replaced or migrated at run-time (e.g., due to a failure or resource insufficiency) by similar components with differently implemented services. The authors in [8] propose to model the variability through adaptation policies, included in the architecture definition. The basic building blocks of adaptation policies are observations and responses. Observations establish information about a system and responses determine modifications in the architecture (additions and removals of architectural elements). The run-time management of policies is addressed by adopting an expert system approach. PLASMA [9] uses an *Architecture Definition Language* (ADL) and a planning-as-model-checking technology to enable dynamic re-planning in the architectural domain. That is, PLASMA uses ADL models and system goals as inputs for generating new plans in response to changes in the system requirements and goals, and in the case of unforeseeable component failures. In contrast to our approach, where we separately model and manage variability related to the robot operation and to the QoS, there is no such separation in any of the previous approaches. Similarly to SmartTCL, PLASMA has the capability of dynamic re-planning. However, the approach presented in [8], as DiaSpec, only uses ECA-based rules.

Although the literature about dynamic variability is quite extensive, we would like to highlight some general-purpose frameworks, like Rainbow[5] and MUSIC[6], that enable self-adaptive system development. As the previous approaches, these frameworks are focused on component-based architectures. Rainbow proposes Stitch [10], a language for expressing adaptation strategies which, among other capabilities, allows representing QoS objectives. Quite similarly to Rainbow, MUSIC manages architecture reconfigurations via goal policies, expressed as utility functions. Each component implementation is associated with some predictors, which precisely specify the impact of a particular implementation on QoS properties. A global utility function computes the overall utility of the application to evaluate different configurations and choose the best one. The idea behind Rainbow and MUSIC is similar to VML but, as [8] and [9], they are too coupled to the underlying architecture. Moreover, they do not enable multiple levels for modeling and managing variability.

In the field of Dynamic Software Product Line (DSPL) [3], MOSKitt4SPL[7] enables designers to model dynamic variability by means of (i) feature models, describing the possible configurations in which the system can evolve, and (ii) resolution models, defining the reconfigurations in terms of feature activation/deactivation associated with a context condition. This specification is automatically transformed into a state machine. Like VML and SmartTCL, this approach enables specification reuse at design-time. However, it does not support expressing optimization of QoS. Also in this line, the DiVA[8] Project provides a tool-supported methodology with an integrated framework for managing dynamic variability. It is worth noting that VML has been inspired by the DiVA DSL [11], which allows managing variability using both ECA rules and optimization of QoS properties. However, DiVA relies on fuzzy logic to capture and describe how systems should adapt. Conversely, VML offers a more precise and less limited way to describe variability, e.g., using mathematical expressions and

[5] Rainbow: www.cs.cmu.edu/~able/research/rainbow
[6] MUSIC: http://ist-music.berlios.de/site/index.html
[7] MOSKitt4SPL: http://www.pros.upv.es/m4spl/features.html
[8] DiVA: http://www.ict-diva.eu/

real variables. This provides designers with a more natural way for describing the variability of their systems, in particular, in some application domains like in robotics. Finally, it is worth noting that both MOSKitt4SPL and DiVA only support *variability in quality*, but not *variability in operation.*

7 Conclusions and Future Work

In this paper we have presented two different DSLs, SmartTCL and VML, which enable robotics software designers to separately specify *variability in operation* and *variability in quality.* Managing the former with SmartTCL provides robustness to contingencies, maintaining a high success rate in task fulfillment. On the other hand, managing *variability in quality* with VML focuses on improving the overall execution performance of the robot under changing situations and limited resources. We have also discussed different possibilities for integrating both variability management mechanisms at run-time in a consistent way, i.e., avoiding conflicting reconfiguration decisions.

Regarding the run-time integration of SmartTCL and VML, we highlight three main contributions: (i) we propose modeling variability at two different levels so that each one has its own (orthogonal) decision space, minimizing the potential conflicts between them (e.g., SmartTCL configures components and VML relies on SmartTCL), (ii) the VML engine, aimed to improve the overall system performance, is not essential and can be eventually stopped, i.e., the system must be fully operative without it, although performing suboptimally, and (iii) SmartTCL and VML enable model reuse, i.e., the same specifications can be used for different platforms or applications (e.g., a TCB to move to a generic location or a VML definition of how to optimize the power consumption can applied in many scenarios).

For the future, we plan to use *Model Driven Engineering* to develop design-time tools for SmartTCL. This would allow creating advanced editors to support developers in their software design and, in particular, to connect both SmartTCL and VML on the meta-model level. This would further reduce the modeling efforts and would better support separate developer roles. We also plan to extend VML with some additional syntax constructs and to improve the supporting tools to provide designers with some advanced model validation and simulation facilities.

Acknowledgements. The project work at University of applied Sciences Ulm is found by ZAFH Servicerobotik[9]. The collaboration is jointly founded by the German Academic Exchange Service (DAAD; Project ID: 54365646; Project leader: Prof. Schlegel) and the General Directorate of International Projects of the Spanish Ministry of Economy and Competitiveness (MINECO; Project leader: Dr. Vicente-Chicote; Project ID: PRI-AIBDE-2011-1094). Juan F. Inglés-Romero thanks Fundación Séneca-CARM for a research grant (Exp. 15561/FPI/10).

[9] http://www.zafh-servicerobotik.de

References

[1] Inglés-Romero, J.F., Lotz, A., Vicente-Chicote, C., Schlegel, C.: Dealing with Run-Time Variability in Service Robotics: Towards a DSL for Non-Functional Properties. In: 3rd International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob 2012), Tsukuba, Japan (November 2012)

[2] Steck, A., Schlegel, C.: Managing execution variants in task coordination by exploiting design-time models at run-time. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), San Francisco, California, USA, pp. 2064–2069 (September 2069)

[3] Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic software product lines. Computer 41(4), 93–95 (2008)

[4] Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.R.: MiniZinc: Towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007)

[5] Bohren, J., Cousins, S.: The smach high-level executive (ros news). IEEE Robotics Automation Magazine 17(4), 18–20 (2010)

[6] Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., Ziane, M.: RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications. In: Noda, I., Ando, N., Brugali, D., Kuffner, J.J. (eds.) SIMPAR 2012. LNCS, vol. 7628, pp. 149–160. Springer, Heidelberg (2012)

[7] Bertran, B., Bruneau, J., Cassou, D., Loriant, N., Balland, E., Consel, C.: DiaSuite: a Tool Suite To Develop Sense/Compute/Control Applications. Science of Computer Programming, Fourth special issue on Experimental Software and Toolkits (2012)

[8] Georgas, J.C., Taylor, R.N.: Policy-based architectural adaptation management: Robotics domain case studies. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 89–108. Springer, Heidelberg (2009)

[9] Tajalli, H., Garcia, J., Edwards, G., Medvidovic, N.: Plasma: a plan-based layered architecture for software model-driven adaptation. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE 2010, pp. 467–476. ACM (2010)

[10] Cheng, S.-W., Garlan, D.: Stitch: A language for architecture-based self-adaptation. Journal of Systems and Software, Special Issue on State of the Art in Self-Adaptive Systems 85(12) (December 2012)

[11] Fleurey, F., Solberg, A.: A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 606–621. Springer, Heidelberg (2009)