Capturing Variability in Adaptation Spaces: A Three-Peaks Approach

Konstantinos Angelopoulos[1], Vítor E. Silva Souza[2], and John Mylopoulos[1]

[1] University of Trento, Italy
[2] Federal University of Espírito Santo, Brazil

Abstract. Variability is essential for adaptive software systems, because it captures the space of alternative adaptations a system is capable of when it needs to adapt. In this work, we propose to capture variability for an adaptation space in terms of a three dimensional model. The first dimension captures requirements through goals and reflects all possible ways of achieving these goals. The second dimension captures supported variations of a system's architectural structure, modeled in terms of connectors and components. The third dimension describes supported system behaviors, by modeling possible sequences for goal fulfillment and task execution. Of course, the three dimensions of a variability model are inter-twined as choices made with respect to one dimension have impact on the other two. Therefore, we propose an incremental design methodology for variability models that keeps the three dimensions aligned and consistent. We illustrate our proposal with a case study involving the meeting scheduling system exemplar.

Keywords: variability, three-peaks, requirements, architecture behaviors

1 Introduction

Adaptive software systems are expected to cope with uncertain environments where requirements cease to be fulfilled now and then. When this happens, an adaptive system needs to reconfigure itself to a different configuration one or more times until fulfillment of its requirements is restored. Possible reconfigurations define an *adaptation space* whose dimensions and size determine the degree of adaptivity of the system-at-hand. Much of the literature on adaptive software systems in the past 15 years has focused on variability that is grounded on requirements or architectures [3, 22, 5, 18, 15, 8, 11]. In this respect, such proposals are limited with respect to adaptivity in that they are blind to dimensions of adaptation other than the native one.

The one and only objective of this paper is to go beyond this one-dimensional view of adaptation spaces by defining adaptation spaces that accommodate three complementary dimensions. The first dimension captures variability in fulfilling requirements and represents variability in the problem part of the adaptation space. The other two dimensions capture variability with respect to behavior and architecture. These dimensions capture variability in the solution space of the system-to-be, representing how, by whom and in what sequence requirements are to be fulfilled. Together, the three dimensions constitute the adaptation space where an adaptive system searches for alternative reconfigurations.

More specifically, we propose a parametrized model for adaptation spaces that is constituted by a requirements, an architectural and a behavioral dimension. Moreover, the paper proposes a technique for building such models by adopting a three-peaks approach where an adaptation space is defined iteratively by introducing some requirements, deciding on their architectural and behavioral dimensions, and then going back and introducing more requirements, including ones that are determined by architectural and behavioral decisions, extending [14]. The requirements dimension is captured by extended goal models in the spirit of [20].

The rest of the paper is structured as follows. Section 2 presents the research baseline of this work. Section 3 proposes a notation for modeling behavioral and structural variability. In Section 4 the three-peaks process is introduced, and it is evaluated in Section 5 with an extended version of the meeting scheduling exemplar. Finally, in Section 6 we discuss and compare related work, while Section 7 concludes.

2 Preliminaries, With Motivating Examples

This section presents the research baseline for this paper.

Goal models. Following our previous work [1], we use goal models to represent requirements. Fig. 1 shows the goal model for a *Meeting Scheduler* system, used as a case study in this paper. The main requirement for the system, *Schedule Meeting*, is represented by the top goal *G0: Schedule Meeting*. AND/OR refinements with traditional Boolean semantics, allow us to refine this goal into finer levels of granularity, down to simple *tasks* that can be operationalized by a component of the system-to-be. Alternatively, *domain assumptions* indicate properties that must hold for the system to work, such as the availability of local and hotel rooms [10].

Even though OR refinements indicate that one of the lower level goals needs to be implemented to satisfy the parent goal, for an adaptive system it is useful to implement all alternatives because this allows multiple reconfigurations during adaptation. Hence, some (in our example, all) OR refinements can be marked as *variation points* (see labels *VP1*−*VP3* in Fig. 1). In this case, all tasks associated with each variation must be implemented and the system can switch from one configuration to another during adaptation [19], as long as it adheres to its *behavior model* (discussed next).

Awareness Requirements (*AwReqs*) [20] impose constraints on the failure of other requirements, and trigger adaptation. They serve as requirements for the monitoring component of the adaptive system's feedback loop. The degree of failure of other requirements is measured by variables named indicators. For example, *AR4* prescribes that *G6* should never fail, whereas *AR3* indicates that

90% of the time *schedules are produced within one day*. Therefore, the restrictions of the associated monitored indicators are $I_4 = 100\%$ and $I_3 \geq 90\%$. If the indicators stray this range, then the associated *AwReq* fails.

Another source of variability along the requirements dimension consists of *control variables*. These represent the amount of resources and effort allocated for the system-to-be while it fulfills its requirements. For instance, *FhM* represents *from how many* participants the system should collect time tables before goal *G5* is considered satisfied (a percentage value). *MCA* is another control variable that represents the *maximum conflicts allowed* for the timeslot chosen for the meeting and participant time tables. *RfM* is yet another, representing how many local (on the premises) rooms have been allocated for meetings, while, *HfM* represents how many hotel rooms are reserved for meetings, and finally *VPA* indicates whether the system has authorization to access personal time tables.

Control variables and variation points, hereafter *requirement control parameters* (*ReqCPs*), can be adjusted at runtime by the adaptation mechanism, to fix failing *AwReqs*. The qualitative relation between *AwReqs* and parameters is captured through a systematic process called *system identification*. During this process the domain expert captures the positive or negative influence that a parameter change can have on an *AwReq*. More specifically, the differential relationship $\Delta(I_2/MCA) < 0$ means that by increasing *MCA* by one unit the success rate of *AR2* will decrease. Similarly, $\Delta(I_5/MCA) > 0$ means that by increasing *MCA* the success rate of *AR5* will increase. Differential relations are symmetric with respect to increases/decreases, meaning that if *MCA* is decreased the success rate of *AR5* will also decrease.

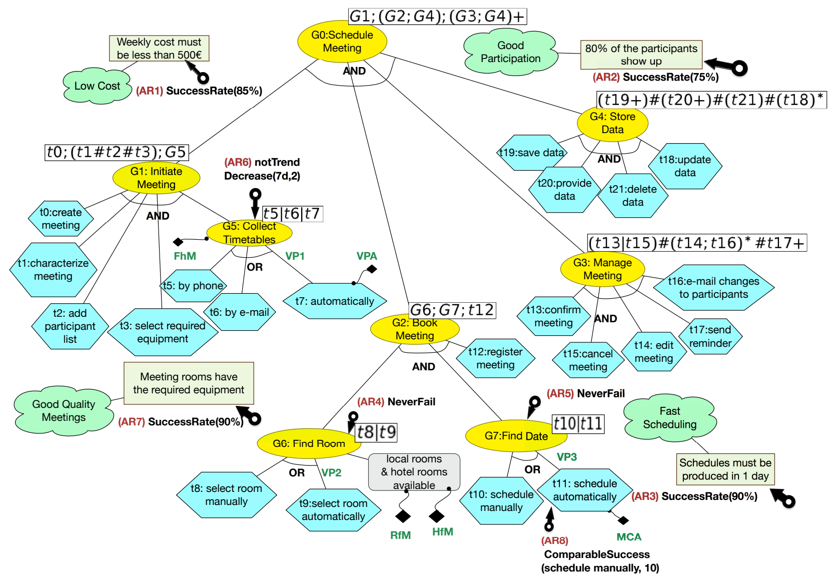


Fig. 1: Goal model for the Meeting Scheduler case study with flow expressions.

Behavioral models. We represent the behavior of the system using *flow expressions* [17, 7] as attachments to each goal (in Fig. 1, goals *G0–G7*). These are extended regular expressions that describe the flow of system behavior, with each atomic component of allowed sequences of fulfillment of sub-goals that lead to the fulfillment of a parent goal.

The operators `;` (sequential), `|` (alternative), `opt()` (optional), `*` (zero or more), `+` (one or more), `#` (shuffle) allow us to specify sequences of system actions that constitute a valid behavior. Shuffle specifies that its operands are to be fulfilled concurrently. For example, *G0* # *G1* means that goals *G0* and *G1* are to be fulfilled in parallel. Of course, each of these goals has its own flow expression to describe in what order its own subgoals and tasks are to be fulfilled/executed.

Some of the operators presented above introduce behavioral variability that cannot be captured with goal models. For example, the meeting scheduler may send a number of reminders to participants before a meeting (*NoR*), thereby lowering the chances any participant will forget, and increasing chances that *Good Participation* will succeed. To this end, behavioral variability introduced behavioral control parameters.

Software architecture. The software architecture of a system constitutes a high-level representation of its structure. It depicts how system components are interconnected and what properties they have. The software architecture is highly coupled to the requirements of a system since the latter prescribes what needs to be achieved and *why*, while the former describes *how* fulfillment is achieved. Architectures are described in terms of the concepts of components and connectors. A component constitutes the basic building block of architecture and is responsible for carrying out operations towards the fulfillment of goals. Components can be software, hardware components or human actors that interact with the system through an interface. Components interact with each other within an architecture using communication links, named connectors. For our purposes, we use class diagrams to represent an architecture [9], where classes model components, while associations model connectors. For example, the class diagram in Fig. 2 shows the architecture of the meeting scheduler system.

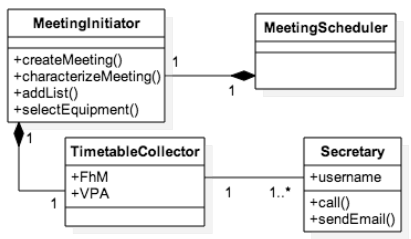


Fig. 2: Architectural diagram for the Meeting Scheduler

Variability is captured in architectural models in terms of alternative components that can fulfill the same goal, but with different qualitative properties (e.g., better performance but lesser usability). Variability here can also be introduced by having a number of component instances participating in the runtime architecture. For instance, the meeting scheduler may have an additional component to what is shown on Fig. 2 that takes in meeting scheduling requests and distributes them among one or more servers each of which consists of the architecture shown in Fig. 2. This kind of variability is exploited by the RAINBOW framework [8].

3 Capturing and Exploring Variability

In the previous section we motivated the need to introduce variability along all three dimensions – goals, behaviors, architecture – to ensure that the system has a large space of adaptation options in trying to cope with one or more requirements failures. In this section, we continue the line of research reported in [19] by demonstrating how to elicit and capture behavioral and architectural *control parameters* (*CPs*) and their impact on system requirements.

Variability in Behavior. The semantics of AND/OR refinements are clear at design time: If goal G is AND/OR refined into sub-goals $G1, ..., Gn$, then the functionality of the system-to-be needs to include functions that fulfill all/at least one of $G1, ..., Gn$.

Behavior talks about the allowable sequences of fulfillment of $G1, ...Gn$ at runtime. Each sequence needs to include one or more of $G1, ..., Gn$, but not all. So, it can be the case that for an AND-refinement we have sequences that fulfill only some of $G1, ..., Gn$ and for OR refinements we have sequences that fulfill all of $G1, ..., Gn$. For example, the goal *Manage Meeting*: although all of its tasks must be implemented, *confirm meeting* and *cancel meeting* are actually conflicting and their use cannot coincide in the same execution sequence. Therefore, the $|$ operator indicates that only one of the two is allowed for any one execution of the system as shown in Fig. 1.

The **;** operator is useful when modeling the behavior of an AND-refinement and prescribes the order in which sub-goals/tasks must be fulfilled. It is common practice in software design to impose only one possible order, thereby limiting the reconfiguration capabilities of the system-to-be. In our framework, the designer is encouraged to select multiple alternative behaviors for fulfilling a goal. Accordingly, we introduce *behavioral control parameters* (*BCPs*) that are assigned to the goal's behavior and whose possible values are all the allowed sequences. A *BCP* is defined as $(|[\mathit{parameter\ name}]alt_1\ ...\ alt_n)$, using infix notation for the alternative operator. For example, for the goal *Book Meeting* if the meeting organizers select a meeting room first and then find a date, participation might be low because of conflicts with participant time tables. If they select the date first and the room afterwards, participation may improve but it is not guaranteed that the selected room will have all required equipment. A *BCP* defined by $(|$ $[BCP1]$ $G6; G7$ $G7; G6)$ takes as values the two possible sequences $G6; G7$ and

$G7; G6$. It's impact on the requirements is captured by the differential relations $\Delta(I_2/BCP1)[G6; G7 \rightarrow G7; G6] > 0$ and $\Delta(I_7/BCP1)[G6; G7 \rightarrow G7; G6] < 0$ while the new behavior for the goal *Book Meeting* is depicted in Fig. 3a.

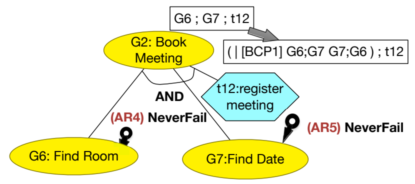
Another variability factor of system behavior is related to the multiplicity of the fulfillments of a goal or a task. When there is the option for the system to fulfill multiple times a goal or a task, the designer must consider the impact of this variability on *AwReqs*. For example, the task *t17 send reminder* is performed by the system–to-be and can be executed multiple times if the goal *Good Participation* is failing. To this end, as depicted in Fig. 3c, we substitute when needed the operators * and + with a *BCP* (in this case named *NoR*) and based on a differential relation such as $\Delta(I_2/NoR) > 0$ the adaptation mechanism can adjust its value when *Good Participation* is failing. The range of values of NoR varies from one to five executions of task *t17*.

The repetitive execution of a task or fulfillment of a goal raises the issue of time synchronization. In the previous example, if $NoR = 3$ and all the reminders are sent one after the other within seconds, the outcome is likely to be an unhappy one. Hence, we introduce a *behavioral function wait()* that takes as argument a *BCP* with a range of values related to time units, in this case days. This function is part of the behavioral model as shown in Fig. 3c and $BCP3$ is defined as $(\mid [BCP3]\ 1day\ 2days\ 3days)$.

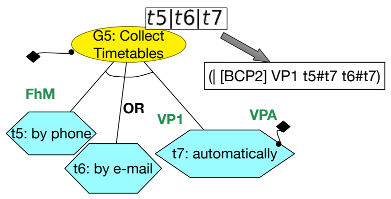
Next, we revisit OR-refinements in order to extract additional variability. The traditional perception of these refinements at runtime is that the satisfaction of any subgoal would lead to the satisfaction of the parent goal. Therefore, the *ReqCPs* associated to an OR-refinement have as candidate value one of the subgoals. The system-to-be though may require in certain occasions the fulfillment of all the subgoals to guarantee the satisfaction of the parent goal. For example, scheduling a meeting requires the fulfillment of the goal *G5: Collect Timetables* that can be achieved by either contacting the participants *by phone*, *by e-mail* or collecting them *automatically* from a common system calendar. However, when one or more of the invited participants do not use the system calendar the third option could harm *AR2*, since these participants will not receive any invitation for the meeting. Dealing with such a situation requires the utilization of all the alternatives under the OR-refinement. This means that participants who do not have an account for using the system's calendar and therefore their timetables must be collected either *by phone* or *by e-mail* while the timetables of the remaining participants can be collected automatically by the system. To capture this additional variability a new *BCP* is introduced defined as $(\mid [BCP2]\ VP1\ t5\#t7\ t6\#t7)$, as depicted in Fig. 3b.

Variability in Architecture. We consider next the third peak, architecture, looking for opportunities to introduce variability. In order to be fulfilled, each goal or task must be assigned to at least one component[3]. For this peak, as we mentioned before, there are two sources of variability. The first is related to each component's multiplicity. Certain components may be instantiated multiple times for requirements to be fulfilled. For example, as shown in Fig. 2, an

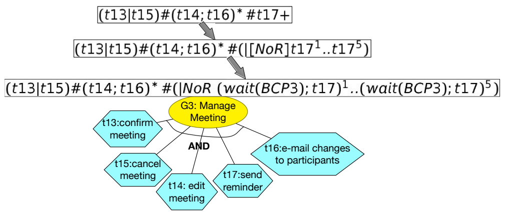
[3] Each component must be able to satisfy on its own the assigned goal.



(a) BCP from AND-refinement



(b) BCP from OR-refinement


(t13|t15)#(t14;t16)* #t17+
(t13|t15)#(t14;t16)* #(|[NoR]t17^1..t17^5)
(t13|t15)#(t14;t16)* #(|NoR (wait(BCP3);t17)^1..(wait(BCP3);t17)^5)
G3: Manage Meeting
AND
t13:confirm meeting
t15:cancel meeting
t14: edit meeting
t17:send reminder
t16:e-mail changes to participants

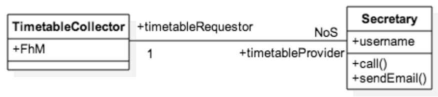
(c) BCP from multiplicity operator

Fig. 3: Behavioral Control Parameters (BCPs) elicitation

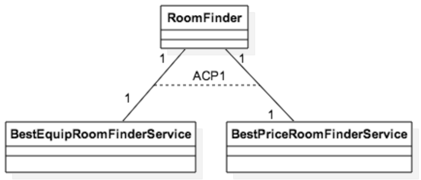
instance of the component *TimetableCollector* can be associated with multiple instances of the component *Secretary*. The number of instances of the latter, is an adjustable variable that affects the operational cost of the meeting scheduling process ($AR1$), but also how fast the meetings are scheduled ($AR3$). We refer to such variables as *architectural control parameters* (*ACPs*) following the same definition construct as *BCPs*. In this case we introduce the number of secretaries NoS parameter defined as (| $[NoS]$ 1..5) that will substitute the abstract multiplicity notation, representing explicitly the presence of a new configuration point. The impact of this ACP on the requirements is captured by the differential relations $\Delta(I_1/NoS) < 0$ and $\Delta(I_3/NoS) > 0$.

The second source of architectural variability is related to the selection among multiple candidate components that are assigned with the same goal/task. For the goal *Find Room* we have two candidate software components that are both part of the system and can be used interchangeably. The first component finds the cheapest room reducing the overall cost of the meetings, but does not guarantee that all the required equipment will be present, while the other one finds the best equipped room but might exceed the budget available for scheduling meetings. These two components can be used either interchangeably or concurrently. The concurrent use of both components allows the users select which result is more suitable for them. In specific occasions such as low budget periods, the system may switch to the exclusive use of the component that provides the best price. Therefore, as shown in Fig. 4b we add to the architecture model an ACP named $ACP1$ with candidate values all the possible uses of these components, with the following definition (| $[ACP1]$ *BestEquipRoomFinder*

$BestPriceRoomFinder\ BestEquipRoomFinder\#BestPriceRoomFinder$). The shuffle operator indicates concurrent use of the operand components.



(a) ACP for component instances



(b) ACP for alternative components

Fig. 4: Architectural Control Parameters (ACPs) elicitation

Variability in the environment. In all previous cases we have seen that factors from the system's environment affect adaptation decisions. Accordingly, our variability model needs to capture variability in the environment and its impact on requirements. Towards this end, we introduce a domain model, as shown in Fig. 5. Environmental variability is captured here with a new type of parameter named *environmental parameter* (*EP*).

An *EP* can indicate the number of instances of a domain entity and therefore its multiplicity in the domain model. The difference from architectural multiplicity is that in the case of the environment the adaptation mechanism has no control on the value of the *EPs*. For instance, there is no control on the *number of meeting requests* (*NoMR*) the meeting organizers are sending, neither the *number of participants* (*NoP*) attending a meeting, and therefore these are represented as *EPs*.

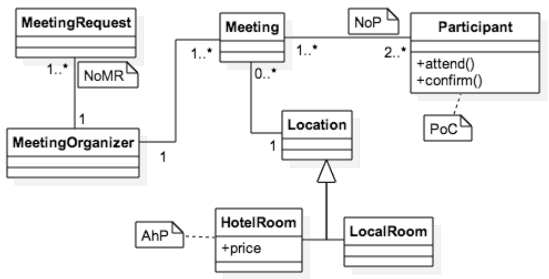


Fig. 5: Domain model for the Meeting Scheduler environment

The attributes and the operations of domain entities constitute another source for environmental variability. For example, participants may confirm their participation, but in the end not attend a meeting. The *EP percentage of con-*

sistency (PoC) captures this, while the *average hotel price* captures the current average cost for reserving a hotel room for meetings.

EPs influence the *AwReqs* in the same manner as *CPs*. However, the adaptation mechanism can only monitor them, identifying undesired situations and change *CPs* to compensate for changes. For example, when the PoC is decreased because participants tend to forget meetings they are supposed to attend and the participation is harmed according to the differential relation $\Delta(I_2/PoC) < 0$, then the adaptation mechanism can increase the *NoR* to compensate.

4 A Three-Peaks Modeling Process

The modeling process for three-peaks models is depicted in Fig. 6. It guides the elicitation of all elements of a three-peaks model, including control parameters. Our process is iterative and intertwined, analyzing and expanding problem and solution spaces simultaneously.

The process starts by getting as input a goal or a task, which initially will be the root goal such as *G0: Schedule Meeting*. The next step is to identify if there are any *AwReqs*, softgoals or domain assumptions related to the input. Then, if the input is a goal, it is refined into subgoals, otherwise the requirement and behavior analysis are skipped. The designers, along with domain experts, examine what needs to be fulfilled and how, starting from eliciting parameters required for that inserted goal to be satisfied, such as how many conflicts are allowed before finding a date or if the system can view private appointments. These parameters are *ReqCPs* and their values may vary during alternative executions of the system.

Continuing the analysis of how a goal can be fulfilled, designers provide an initial behavioral model using the notation introduced in Section 2. If the goal is OR-refined then each subgoal becomes a candidate value for a *ReqCP* such as $V1 - V3$ in Fig. 1. Then, the behavioral model is refined by adding a BCP with range of values according to existing *ReqCP* and shuffle combinations of the refinements as in Fig. 3b. In the case of AND-refinement, the order in which the operands of sequential behaviors (the parts of the model that include only the ; operator) is examined. If a different order of the operands implies influence to different *AwReqs* a new BCP is introduced with range of values, all the potential orders. Concluding this iteration of behavior analysis, the process examines every $*$ and $+$ operators in order to substitute them with a BCP, if needed, as described in Section 3. In that case also the *wait(BCP)* function with its own BCP is added. The last step leads to a new refinement of the goal since a *wait* task is added as a refinement of the examined goal.

Moving to the architecture peak, designers associate the input goal or task to one or more components of the architecture. This determines who is responsible for the satisfaction of the goal or task. When more than one component is assigned, an ACP is added and can be tuned by the adaptation mechanism at runtime in order to activate the most suitable component or a combination of them for fixing failing requirements. Next, if the new component can be in-

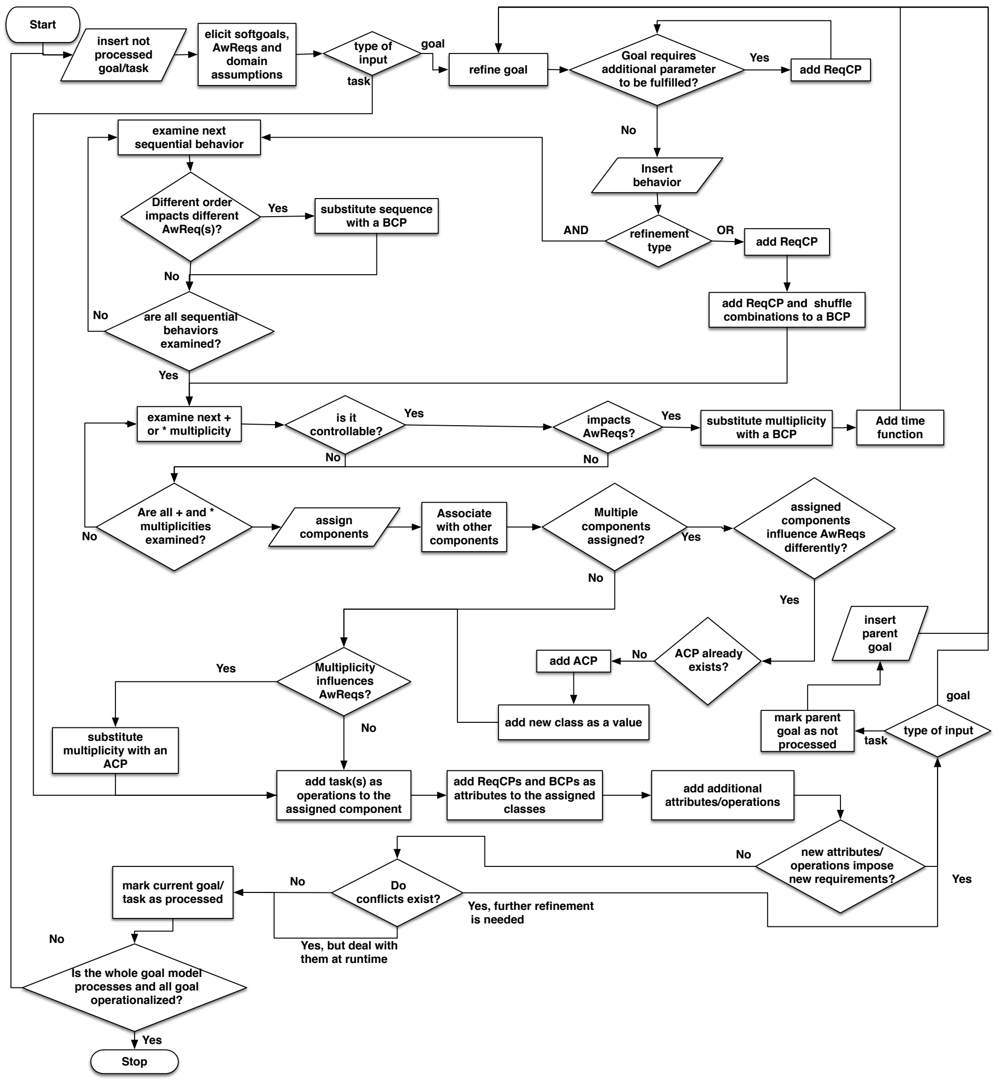

Start
insert not processed goal/task
elicit softgoals, AwReqs and domain assumptions
type of input
goal
task
refine goal
Goal requires additional parameter to be fulfilled?
Yes
add ReqCP
No
Insert behavior
refinement type
AND
OR
add ReqCP
add ReqCP and shuffle combinations to a BCP
examine next sequential behavior
Different order impacts different AwReq(s)?
Yes
substitute sequence with a BCP
No
are all sequential behaviors examined?
No
Yes
examine next + or * multiplicity
is it controllable?
Yes
No
impacts AwReqs?
Yes
No
substitute multiplicity with a BCP
Add time function
Are all + and * multiplicities examined?
No
assign components
Associate with other components
Multiple components assigned?
Yes
No
assigned components influence AwReqs differently?
Yes
ACP already exists?
No
add ACP
add new class as a value
insert parent goal
mark parent goal as not processed
type of input
goal
task
Multiplicity influences AwReqs?
Yes
No
substitute multiplicity with an ACP
add task(s) as operations to the assigned component
add ReqCPs and BCPs as attributes to the assigned classes
add additional attributes/operations
new attributes/ operations impose new requirements?
No
Yes
Do conflicts exist?
No
Yes, further refinement is needed
Yes, but deal with them at runtime
mark current goal/ task as processed
No
Is the whole goal model processes and all goal operationalized?
Yes
Stop

Fig. 6: The three-peaks process as a flowchart

stantiated multiple times at runtime and this number has impact on the *AwReqs* while under the control of the adaptation mechanism, the associated multiplicity is substituted with an ACP. Then, the assigned components get as attributes the *ReqCPs* and *BCPs* of the goal, as they must be aware of what behavior must follow and what are the values of these parameters. Once the previously elicited variability has been embedded in the assigned component, the designers of the architecture, provided that the goal is fully or partially operationalized, add the tasks produced by the refinement as operations. Finally, the designers may provide additional attributes and operations to the component of more technical nature that are not related neither to requirements nor to behavior. For every new attribute or operation, the process must investigate whether there is need of adding new requirements. If the initial input was a goal then it is refined again, but in case of a task then it is the parent goal that must be processed again.

The last step of the process inspects if the current set of configurations is able to guarantee the the satisfaction of all the *AwReqs* related to the investigated goal under any possible environmental condition. In case there are situations where the system is not able to guarantee success of all the related *AwReqs*, then two actions can be taken: a) perform further refinements, finding new *CPs*, goals or tasks; or b) deal with conflicting requirements, using the conflict resolution mechanism of our previous work [1]. When all goals and tasks are processed and every goal is operationalized, the process terminates.

5 Evaluation

Following the three-peaks process presented in the previous section we produce a goal model with annotated behavior (Fig. 7) and an architectural model (Fig. 8) which includes several additional parameters over what was presented in Section 2.

More specifically, six new *CPs* are derived from the behavioral model ($BCP1$–$BCP5$ and *NoR*) and two from the architectural model ($ACP1$ and *NoS*). Moreover, the three-peaks process resulted in eliciting three additional tasks ($t22$, $t23$ and $t24$). The task *t23: be online* along with $AR9$ are result of the attribute assigned to the component *Database* that is responsible for the goal *G4: Store Data*. This prescribes that the status of the *Database* must be monitored to ensure that it is constantly online. The task *t24:wait* is introduced by the assigned behavior to goal *G3: Manage Meeting*. Finally, the task *t22: do meeting online*, has been introduced to resolve situations where there are few suitable dates due to many conflicts among participants, and there are not enough available rooms.

In our previous work [1] we did not take into account the holding conditions of the environment. This carries the risk of choosing the wrong adaptation for fixing failing requirements. For instance, consider the case where participants forget to attend their meetings, resulting in the failure of $AR2$. The adaptations offered by the original goal model (not generated by the three-peaks process) for fixing $AR2$ are either to start viewing private appointments of participants by setting *VPA* to true or decreasing *MCA* allowing fewer conflicts. Neither one

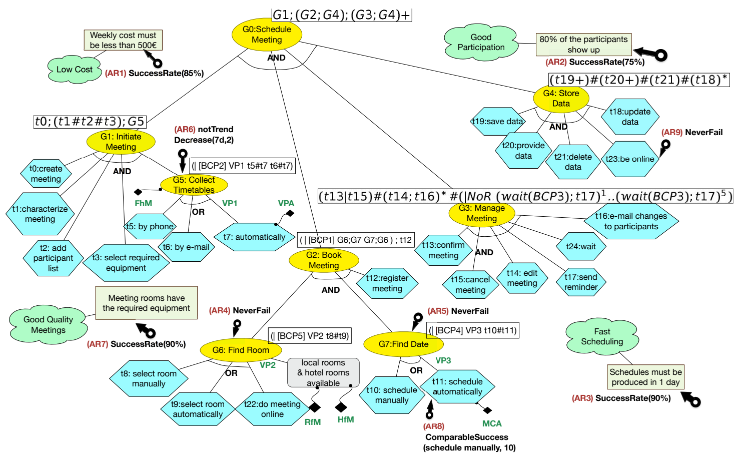


Fig. 7: The goal model after the three-peaks process

anticipates the real cause of the failure. The three-peaks model though offers the parameter NoR that increases the number of reminders thereby tackling the source of the problem, and capable of increasing the success rate of $AR4$.

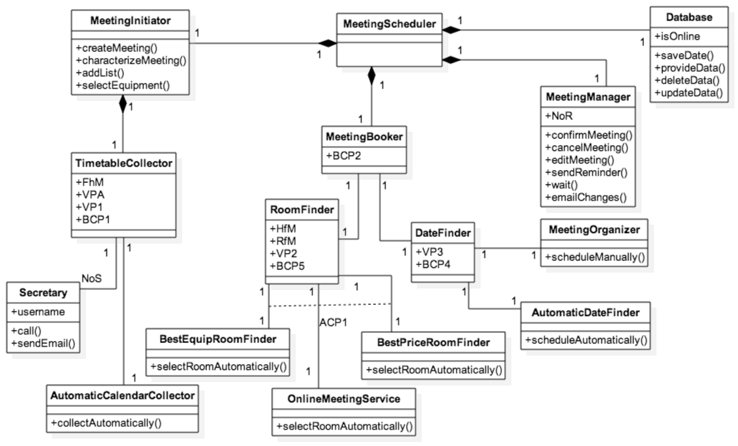


Fig. 8: The architecture model after the three-peaks process

Another case where requirements-only variability proves to be insufficient concerns the room selection by the meeting organizers before or after finding a date for their meeting. Each of the alternative orders works well in different contexts. Selecting room first guarantees good quality meetings, since the meeting organizers select a room that can provide all the required equipment, assuming that the invited participants are available the same date the room is available, otherwise the success rate of $AR5$ is at risk. On the other hand, when meeting organizers select date first, it is more likely that they will find a date convenient to most invited participants, but a sufficiently equipped room might not be found in periods with high workload for the meeting scheduler, decreasing the success rate of $AR7$. Using behavioral variability, an adaptive meeting scheduler executes the order that complies with the existing context by tuning $BCP1$. Moreover, to maintain the equilibrium between the success rate of $AR1$ and $AR7$ when the system selects rooms automatically the system can use either a component that finds the cheapest room available or the best equipped respectively, exploiting architectural variability and more particularly $ACP1$.

The previous failure scenarios show that the high variability models of the three-peaks process can handle better changes in the system's environment where the requirements-only model would provide ineffective adaptations. A limitation of our approach is that dependencies among *CPs* are not captured. For instance, it makes sense for MCA to be changed only if the value of $VP3$ is set first to "schedule automatically". In order to alleviate this obstacle, we are planning to extend our notation in order to capture this kind of constraint. Another limitation on the scalability of our proposal is that for every variable introduced into the model, its impact on all *AwReqs* must be examined.

6 Related Work

Having as a starting point a goal model, Yu et al. [23] propose heuristics to derive other models such as feature models, statecharts and component-connector models. Their purpose is to express the same level of variability in different dimensions of the system. On the other hand, our approach intends to capture the interaction of the system's dimensions and incrementally elicit additional variability along each one of them.

The STREAM-A approach presented in [16] derives ACME architectural models from goal models using model transformations. The environment's influence on the requirements is captured in terms of context. The main purpose of this work is to relate the requirements to components and place accordingly the actuators and the sensors of the adaptation mechanism. While this approach provides a method for binding requirements and architecture, it overlooks the factor of behavior in the adaptation process. In comparison to our proposal, there is no effort of exploring and handling the variability of the derived model.

In the work of Kramer and Magee [12] goals and components are related with reactive plans. When a failure takes place or a goal is changing, the proposed adaptation mechanism generates a new plan of actions that needs to be carried out and the available components that are required are reconfigured to the current architecture. This approach, as well as ours, demonstrates the advantages of architectural variability, by assigning goals to multiple components. However, our work goes a step further by modelling the impact that every alternative has on a goal's satisfaction.

Chen et al. [4] combine requirements and architectural adaptations. The stakeholders state their preferences about the goals and clarify what expectations they have for their satisfaction. Then the adaptation framework monitors the system and if expectations are not met it attempts to find a different architectural solution. In case the problem is still not solved, a new specification is derived from the goal model attempting a reconfiguration at requirements level. This approach doesn't take into account the variability of the behavior which we express by flow expressions. Furthermore, the priority is given to architectural alternatives while in our work all the dimensions are candidates for offering solutions.

Other approaches that derive architectures from goal models such as the work of Chung et al. [6] and Lamsweerde [21], offer systematic methods for relating goals, behaviors and architectures. Even though there is no notion of variability for adaptation purposes within these approaches, they can by exploited to elaborate the production of the three-peaks model as proposed here.

In [13], Lapouchnian et al. describe how to derive high variability business process models from goal models and how softgoals can guide the reconfiguration of a business process. Our approach follows the same line of research, adding a richer notation and capturing behavioral variability that goes beyond the OR-refinements of the goal models. Moreover, we introduce environmental variables that can influence the satisfaction of our goals and drive the reconfiguration process.

Finally another variability management approach that is related to our work is Dynamic Software Product Lines (DSPLs). In [2] DSPLs are applied to service-based system for adapting at runtime to the user's requirements by adding and removing features and reconfiguring the business process in order to support these changes. Our work examines architectural variability at a component level, where different components can satisfy the same goal(s) but may influence different indicators. The reason is that the components share common features, while they differ to some others. Therefore, DSPLs could be complementary to our work to express at a deeper level of detail our architectural variability.

7 Conclusions and Future Work

We propose a systematic process for extracting incrementally variability from goal models. The source of variability lies in the three peaks of a software system: requirements, behavior and architecture. We investigate how variability can be elicited along each peak, introducing behavioral and architecture control parameters and how to model environmental variability. We also present a three-peaks process to derive incrementally high variability requirements, behavioral and architecture models. Finally, we have evaluated our models through execution scenarios of the meeting scheduler exemplar, showing that offering adaptations along three peaks enables the system to handle more failures.

Our future research includes the implementation of a tool that will support the design of our three-peaks models, as well as further evaluation with real case studies. We also plan to experiment with a simulation of the meeting-scheduler exemplar using the three-peaks model extending our previous work [1] on handling multiple failures by exploring the quantitative form the differential relations and applying control theoretical adaptation techniques for multivariable system with multiple inputs and multiple outputs.

Acknowledgment

This work has been supported by the ERC advanced grant 267856 "Lucretius: Foundations for Software Evolution" (April 2011 – March 2016, http://www.lucretius.eu) and Brazilian research agencies CAPES and CNPq (process numbers 402991/2012-5, 485368/2013-7 and 461777/2014-2).

References

1. K. Angelopoulos, V. E. S. Souza, and J. Mylopoulos. Dealing with multiple failures in zanshin: A control-theoretic approach. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pages 165–174, New York, NY, USA, 2014. ACM.
2. L. Baresi, S. Guinea, and L. Pasquale. Service-oriented dynamic software product lines. *Computer*, 45(10):42–48, Oct 2012.
3. L. Baresi, L. Pasquale, and P. Spoletini. Fuzzy Goals for Requirements-driven Adaptation. In *Proc. of the 18$^{\text{th}}$ IEEE International Requirements Engineering Conference*, pages 125–134. IEEE, 2010.
4. B. Chen, X. Peng, Y. Yu, B. Nuseibeh, and W. Zhao. Self-adaptation through incremental generative model transformations at runtime. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 676–687, New York, NY, USA, 2014. ACM.
5. B. H. C. Cheng, P. Sawyer, N. Bencomo, and J. Whittle. A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. In A. Schürr and B. Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 468–483. Springer, 2009.
6. L. Chung, S. Supakkul, N. Subramanian, J. Garrido, M. Noguera, M. Hurtado, M. Rodríguez, and K. Benghazi. Goal-oriented software architecting. In P. Avgeriou, J. Grundy, J. G. Hall, P. Lago, and I. Mistrík, editors, *Relating Software Requirements and Architectures*, pages 91–109. Springer Berlin Heidelberg, 2011.
7. F. Dalpiaz, A. Borgida, J. Horkoff, and J. Mylopoulos. Runtime goal models. In *Proc. of the IEEE 7$^{\text{th}}$ International Conference on Research Challenges in Information Science*, pages 1–11. IEEE, 2013.
8. D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, 37(10):46–54, 2004.
9. J. Ivers, P. Clements, D. Garlan, R. Nord, B. Schmerl, and J. R. Silva. Documenting component and connector views with uml 2.0. Technical report, DTIC Document, 2004.

10. I. Jureta, J. Mylopoulos, and S. Faulkner. Revisiting the Core Ontology and Problem in Requirements Engineering. In *Proc. of the 16*$^{\text{th}}$ *IEEE International Requirements Engineering Conference*, pages 71–80. IEEE, 2008.
11. J. Kramer and J. Magee. Self-Managed Systems: an Architectural Challenge. In *Future of Software Engineering (FOSE '07)*, pages 259–268. IEEE, 2007.
12. J. Kramer and J. Magee. A rigorous architectural approach to adaptive software engineering. *J. Comput. Sci. Technol.*, 24(2):183–188, 2009.
13. A. Lapouchnian, Y. Yu, and J. Mylopoulos. Requirements-driven design and configuration management of business processes. In *In BPM (2007)*, pages 246–261, 2007.
14. B. Nuseibeh. Weaving together requirements and architectures. *Computer*, 34(3):115–119, Mar 2001.
15. P. Oreizy et al. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.
16. J. Pimentel, M. Lucena, J. Castro, C. Silva, E. Santos, and F. Alencar. Deriving software architectural models from requirements models for adaptive systems: the stream-a approach. *Requirements Engineering*, 17(4):259–281, 2012.
17. J. a. Pimentel, J. Castro, J. Mylopoulos, K. Angelopoulos, and V. E. S. Souza. From requirements to statecharts via design refinement. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 995–1000, New York, NY, USA, 2014. ACM.
18. V. E. S. Souza. *Requirements-based Software System Adaptation.* Phd thesis, University of Trento, Italy, 2012.
19. V. E. S. Souza, A. Lapouchnian, and J. Mylopoulos. System Identification for Adaptive Software Systems: a Requirements Engineering Perspective. In M. Jeusfeld, L. Delcambre, and T.-W. Ling, editors, *Conceptual Modeling – ER 2011*, volume 6998 of *Lecture Notes in Computer Science*, pages 346–361. Springer, 2011.
20. V. E. S. Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos. Awareness Requirements. In R. Lemos, H. Giese, H. A. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 133–161. Springer, 2013.
21. A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications.* Wiley, 2009.
22. J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *Proceedings of the 2009 17th IEEE International Requirements Engineering Conference, RE*, RE '09, pages 79–88, Washington, DC, USA, 2009. IEEE Computer Society.
23. Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos, and J. C. S. P. Leite. From goals to high-variability software design. In *Proceedings of the 17th International Conference on Foundations of Intelligent Systems*, ISMIS'08, pages 1–16, Berlin, Heidelberg, 2008. Springer-Verlag.