Requirements Evolution: From Assumptions to Reality

Raian Ali[1,2] and Fabiano Dalpiaz[1], Paolo Giorgini[1], and Vítor E. Silva Souza[1]

[1] DISI, University of Trento, Italy
[2] LERO - The Irish Software Engineering Research Centre
{raian.ali,dalpiaz,paolo.giorgini,vitorsouza}@disi.unitn.it

Abstract. Requirements evolution is a main driver for systems evolution. Traditionally, requirements evolution is associated to changes in the users' needs and environments. In this paper, we explore another cause for requirements evolution: *assumptions*. Requirements engineers often make assumptions stating, for example, that satisfying certain sub-requirements and/or correctly executing certain system functionalities would lead to reach a certain requirement. However, assumptions might be, or eventually become, invalid. We outline an approach to monitor, at runtime, the assumptions in a requirements model and to evolve the model to reflect the validity level of such assumptions. We introduce two types of requirements evolution: autonomic (which evolves the priorities of system alternatives based on their success/failure in meeting requirements) and designer-supported (which detects loci in the requirements model containing invalid assumptions and recommends designers to take evolutionary actions).

Key words: Requirements Engineering, Requirements Evolution, Contextual Requirements, Requirements at Runtime

1 Introduction

The satisfaction of users' requirements through a developed system is inherently uncertain. Indeed, requirements evolve, and the original system might become inadequate to meet the evolved requirements. Since such evolution is unavoidable, system necessarily has to evolve in order to keep requirements satisfied. Traditionally (e.g., [1, 2]), requirements evolution is driven by changes in the users' needs, the operational environment (laws, policies, economical situation), the co-operative systems, and the underlying technology. We explore a new and primitive driver for requirements evolution which is the uncertain validity of the assumptions included in a requirements model.

Requirements are expressed via requirements models. These models contain assumptions, rather than certainties, made by designers about the relation between the system, the requirements, and the environment where the system is to operate. The model may state that a certain requirement will be met by correctly developing and executing specific software functionalities and/or by meeting other sub-requirements. Such assumptions could turn out to be invalid when the system operates. The operation could reveal that some assumptions were initially, or eventually become, invalid. The detection of invalid assumptions necessitates evolutionary actions, which result in a revision of the requirements model to reflect reality. Desirably, these actions are done by the system autonomously. However, the designers' intervention could be often required.

Our goal in this paper is to enable monitoring the runtime system operation and exploiting it to evolve the requirements models in a lifelong style. We intend to develop systems that support requirements evolution either autonomously or by recommending designers to take evolutionary actions. For example, a shopping mall adminstration intends to build a system to interact with customers' via their PDAs in order to meet the requirement R= "customers head to cash desk when closing time is approaching":

- *Assumptions*. An analyst develops a requirements model stating that R is reached if "customer is notified to leave" (R_1) and "customer is instructed how to leave" (R_2). The model states also that R_2 is reached by one of two software variants: "digital map is shown on customer's PDA" ($SV_{2.1}$) and "customer is tracked and given voice commands via his PDA" ($SV_{2.2}$). These are just assumptions made by the analyst.
- *Autonomic evolution*. As software is deployed, the analyst assumes that $SV_{2.1}$ and $SV_{2.2}$ are equally able to meet R_2. After two months, the operation reveals that $SV_{2.2}$ succeeded in meeting R_2 less often than $SV_{2.1}$. Thus, software should autonomously evolve the requirements model by giving $SV_{2.1}$ higher priority than $SV_{2.2}$ for R_2.
- *Designer-supported evolution*. Software operation could also reveal that reaching R_1 and R_2 does not always lead to reach R. Customers, even if notified and guided on how to leave, still don't leave the mall on time. If such assumption is often invalid, software will ask designers to revise the requirements model and fix it.

We focus on the evolution of contextual requirements models which capture the relation between the state of the environment where the system operates (context [3]) and requirements. Some contexts activate a requirement and others represent preconditions for applying software variants aiming to meet certain requirement. Recently, several contextual requirements models have been proposed to capture such a relationship [4, 5, 6]. However, modeling contextual requirements is a hard task in which designers need to make assumptions with high uncertainty, such as stating that executing a certain software variant in a specific context will lead to reach a certain requirement.

In this paper, we discuss the evolution of contextual requirements. We articulate the problem of requirements evolution that originates from the invalidity of the assumptions included in contextual requirements (Sec. 2) represented via contextual goal models [5, 7] (Sec. 3). We address the two kinds of evolution introduced earlier (autonomic and designer-supported), specifying what information the system has to monitor at runtime and showing how to use this information for evolving the contextual requirements model (Sec. 4). We end the paper with conclusions and future work directions (Sec.5).

2 Requirements Evolution: a Viewpoint

Software systems operate in an environment. The state of such environment, denoted by the notion of *context* [3], is variable. There is a strong mutual influence between context and requirements. This is particularly true in emerging computing paradigms such as ubiquitous and mobile computing, where context-awareness is fundamental for successful software operation. On the one hand, a certain context might *activate* a requirement or be a *required* precondition for the execution of a software variant designed to reach activated requirements. On the other hand, a requirement corresponds to a *target* context, i.e. the desired state of the environment associated to requirement satisfaction.

In Fig. 1a we depict our general picture about the relation between requirements, software, and context. In Fig. 1b we exemplify it through one specific requirement for a mobile software meant to advertise products to customers in shopping malls. A requirement R is activated—software has to meet it—if its activation context holds. In turn, a context holds if one of its variants holds. A context variant is a conjunction of atomic environmental facts the system can verify. Thus, R is activated if any activation context variant ACV_i is true. R is reached if any of its target context variants TCV_j holds. In order to reach a requirement, a software variant SV_k should be executed. A certain variant is adoptable if any of its required context variants $RCV_{k.j}$ holds.

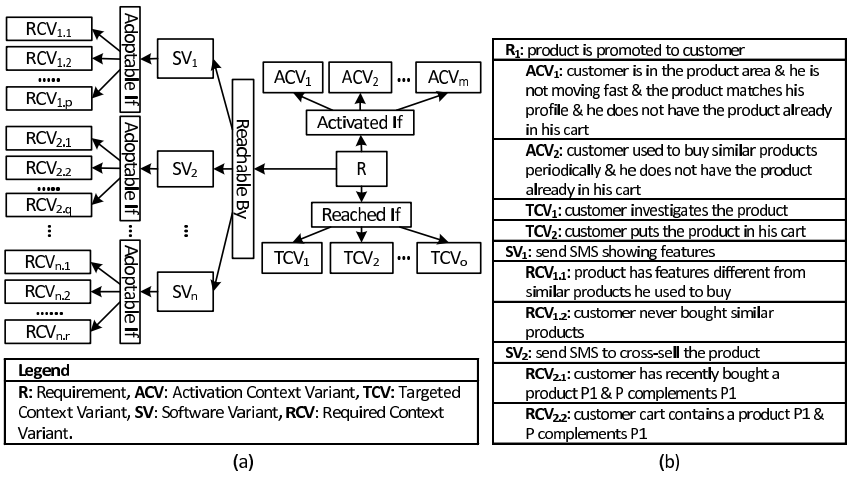


Fig. 1. The relation between Requirements, Context, and Software.

Requirements models contain statements that might be, or become, invalid in practice. Thus, a requirements model contains a set of assumptions. These assumptions should be continuously monitored and, if invalid, requirements models should evolve to reflect reality. We list now some types of assumptions that a requirements model could contain. These assumption types are illustrated via examples taken from Fig. 1b:

1. *Activation assumption*. It concerns the hypothesis that a certain context variant activates a requirement. In practice, a requirement might not need to be activated when one of its activation context variants holds. For example, ACV_1 is presumed sufficient to activate R_1, while in practice ACV_1 could miss some additional contextual conditions, and the promotion might lead to a negative customer reaction if those missing conditions are not considered.
2. *Adoptability assumption*. It concerns the hypothesis that a requirement is met by a software variant which is adoptable in a certain set of required context variants. However, a certain software variant might fail to meet the requirement no matter if one of its required context variants holds. Also, the ability of a software variant to meet a requirement could vary according to each of its required context variants. For example, SV_2 (when $RCV_{2.2}$ holds) could lead to meet R_1 more often than SV_1 (in both of its required context variants $RCV_{1.2}$ and $RCV_{1.1}$).

3. *Refinement assumptions*. It concerns the hypotheses related to the requirements refinements stating that (i) a decomposed requirement is met if all sub-requirements are met; and (ii) a specialized requirement is met if any sub-requirement is met. Suppose R_1 is decomposed into "promote by PDA" ($R_{1.1}$) and "a staff is available for further information" ($R_{1.2}$). Meeting both $R_{1.1}$ and $R_{1.2}$ may not lead to a successful promotion (R_1 is not met). Suppose now a requirements model where R_1 is specialized into "PDA-based promotion" ($R_{1.1'}$) and "staff-based promotion" ($R_{1.2'}$). Meeting $R_{1.1'}$ (e.g. the customer reads the information sent to his PDA) might not lead to a successful promotion (R_1 is not met).
4. *Requirements achievement assumptions*. It concerns the hypothesis that a requirement is satisfied if any of its target context variants holds. In practice, reaching one of these variants may not imply that the requirement is really met. For example, upon executing SV_1 or SV_2, a customer may investigate the product (TCV_1 holds), but this does not necessarily mean he becomes interested in the product.

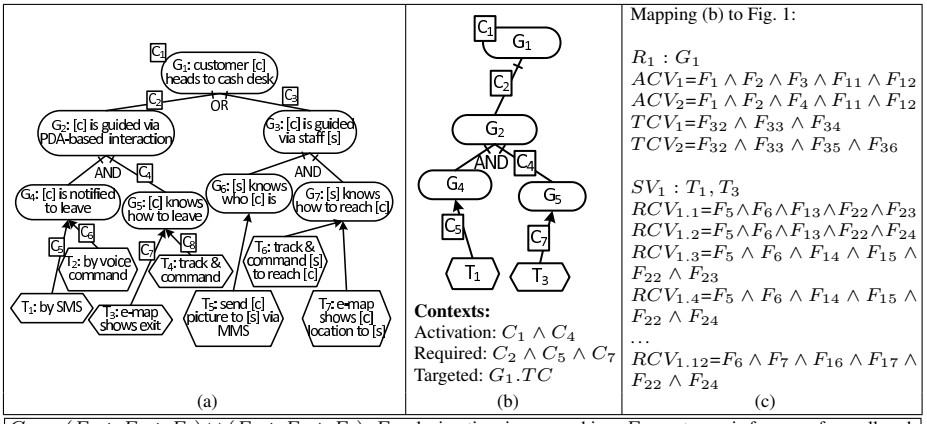
We view *requirements evolution as a continuous movement from assumptions-based requirement to reality-based ones*. The system has to continuously monitor assumptions at runtime and, when an invalid assumption is identified, it is fixed by evolving the requirements model in one of 2 styles; autonomous or designer-supported. Out of the possible combinations between the 4 assumption types and the 2 evolution styles, explained above, we explore 2 combinations; (i) autonomic evolution of adoptability assumptions and (ii) designer-supported evolution of refinement assumptions.

3 Background: Contextual Goal Model

Goal models provide a systematic refinement of user requirements, understood as goals, to derive alternative sets of functionalities software has to support [8, 9]. Goals (graphically represented by ovals) can be refined via AND-decomposition or OR-decomposition. In an AND-decomposition, all subgoals should be achieved to reach the decomposed goal. In an OR-decomposition, the achievement of one subgoal is enough to reach the decomposed goal. Goals are ultimately reached by means of executable processes called tasks (represented by hexagons).

Contextual goal models weave together the variabilities of both goal achievement and context [5, 7]. Context is specified at a set of goal model variation points. The semantics of context influence differs according to each point. The contexts specified at the variation points Root-goal and AND-decomposition are *activation contexts*; they represent stimulating conditions for goals/tasks. The contexts specified at OR-decomposition, Means-end, and Delegation are *required contexts*; they represent adoptability preconditions for alternatives means to reach/execute goals/tasks.

In this work, we also interpret the satisfaction criteria of a goal as a *target context*, i.e., a state of the world to reach. In Fig. 2a, we depict a contextual goal model example. We have refined the contexts $C_1 \ldots C_8$ and $G_1.TC$ using our context analysis technique (for details see [5, 7]), which led to the specifications shown in the table in the bottom of the figure. In Fig. 2b, we show a contextual goal model variant together with its activation and required contexts constructed by accumulating the individual contexts specified at the variation points of each type. Finally, in Fig. 2c, we map the variant shown in Fig. 2b to our view explained in Sec. 2.



$C_1 = (F_1 \wedge F_2 \wedge F_3) \vee (F_1 \wedge F_2 \wedge F_4)$. F_1: closing time is approaching, F_2: customer is far away from all cash desks, F_3: customer is still shopping, F_4: customer is heading far from cash desk
$C_2 = (F_5 \wedge F_6) \vee (F_6 \wedge F_7)$. F_5: customer is using his PDA for different services from time to time recently, F_6: his language is supported currently, F_7: he is familiar with PDA as he stated in the registration form
$C_3 = (F_8 \wedge F_9 \wedge F_{10})$. F_8: staff is close to customer, F_9: staff is free, F_{10}: staff knows a language common to customer
$C_4 = (F_{11} \wedge F_{12})$. F_{11}: customer is visiting the mall for the first times, F_{12}: customer is not familiar with similar mall branches architectures
$C_5 = (F_{13}) \vee (F_{14} \wedge F_{15}) \vee (F_{16} \wedge F_{17})$. F_{13}: customer is often doing some actions on his PDA, F_{14}: customer PDA is not on silent, F_{15}: customer is in a place where ring tone is noticeable with respect to noise level, F_{16}: customer is holding PDA with him, F_{17}: customer PDA vibration is turned on
$C_6 = (F_{18} \wedge F_{19}) \vee (F_{20} \wedge F_{21})$. F_{18}: customer is putting headphones, F_{19}: customer is not listening to loud music/voice on headphones, F_{20}: customer is not making a phone call, F_{21}: customer is not talking to someone
$C_7 = (F_{22} \wedge F_{23}) \vee (F_{22} \wedge F_{24})$. F_{22}: customer has good expertise in navigating e-map, F_{23}: customer's PDA screen is wide enough, F_{24}: the route to a cash desk is simple
$C_8 = (F_{18} \wedge F_{19} \wedge F_{20} \wedge F_{21} \wedge F_{25})$. F_{25}: customer is familiar with direction instructions

$G_1.TC = (F_{26} \wedge F_{27} \wedge F_{28}) \vee (F_{26} \wedge F_{27} \wedge F_{29} \wedge F_{30})$. F_{26}: customer is heading closer to one of the cash desks, F_{27}: customer is not stopping much while heading, F_{28}: customer has not picked a cart, F_{29}: customer has picked a trolley, F_{30}: customer cart is with him.

Fig. 2. An example of a Contextual Goal Model

4 Evolving Contextual Goal Models

We detail now our viewpoint on contextual requirements evolution by showing how contextual goal models are subject to evolution. First, we describe how to monitor the execution of contextual goal model variants (Sec. 4.1). Second, we explain the basic mechanisms to enact autonomic evolution of adoptability assumptions (Sec. 4.2) and designer-supported evolution of refinement assumptions (Sec. 4.3).

4.1 Monitoring Contextual Goal Models

Monitoring requirements is an essential activity to identify the necessity to evolve the system. Evolution is needed whenever some requirements assumptions prove to be invalid in practice. Monitoring means keeping track of the execution of each software variants and the impact it has on requirements (i.e., are requirements met?). Specifically, we focus on requirements expressed in terms of goals in a contextual goal model. Table 1 exemplifies monitoring of adoptability and refinement assumptions for Fig. 2.

Operation	Enacted Variant	RCV					G_4
		A_1	A_2	A_3	B_1	B_2	
I_1	SV_A	T	F	F	F	T	×
I_2	SV_A	F	T	F	T	T	×
I_3	SV_B	F	T	F	T	T	✓
I_4	SV_A	F	F	T	T	F	✓
I_5	SV_B	F	T	T	F	T	✓
I_6	SV_A	T	T	F	F	T	×

(a)

Operation	G_4	G_5	G_2	G_6	G_7	G_3	G_1
J_1	✓	✓	✓				✓
J_2	×	✓	×				×
J_3	✓	✓	×				×
J_4	✓	✓	✓				×
J_5				✓	✓	✓	×
J_6	✓	×	✓				✓
J_7				✓	✓	✓	✓

(b)

Table 1. Monitoring assumptions: (a) adoptability (b) refinement.

In Table 1a, we consider adoptability assumptions taking only the goal G_4 from Fig. 2 (due to space limitations we did not choose the root goal). The goal has two variants; V_A=$\{T_1\}$ and V_B= $\{T_2\}$. Both variants are means to achieve G_4= "customer is notified to leave". There are five required context variants: $RCV_{A_1} = F_{13}$, $RCV_{A_2} = F_{14} \wedge F_{15}$, and $RCV_{A_3} = F_{16} \wedge F_{17}$ for V_A, $RCV_{B_1} = F_{18} \wedge F_{19}$ and $RCV_{B_2} = F_{20} \wedge F_{21}$ for V_B. Every row represents the data collected—via monitoring—during the operation of a specific variant. The columns in the table are an identifier for the operation, an identifier for the enacted variant, the validity of the required context variants, and the satisfaction of the goal the variant should achieve.

In Table 1b, we show refinement assumptions monitoring for the goals in Fig. 2. In line with the characterization of requirements we gave in Fig. 1, every goal has a target context that interprets its satisfaction criteria concretely. Monitoring refinement assumptions means monitoring if the target contexts for root and intermediate goals are reached or not in each operation. In the table, the first column is an identifier for each operation, whereas the following columns reflect the satisfaction of the goals.

4.2 Autonomic Evolution of Adoptability Assumptions

Traditionally, software selects the variant to achieve its current goals based on the policies defined by its designers. However, the variant the system would choose according to such policies might include adoptability assumptions that the operation experience proved to be invalid. When this is the case, the system can adjust its behaviour autonomically—without human intervention—and choose an alternative variant that contains adoptability assumptions proven more valid in the current context. In a contextual goal model, suppose that the root goal is activated (and some subgoals as well), and there exist more than one adoptable goal model variant (hereafter *GMV*) for meeting the root goal. In autonomous evolution, the selection between *GMV*s is based on the operation experience the system has. The system will use such experience (exemplified in Table 1a) and select the *GMV* that demonstrated to be the most successful.

In the following, the decision taken by the system is based on the history of each GMV_i in each of its required context variants ($RCV_{i.j}$). Each pair $\langle GMV_i, RCV_{i.j} \rangle$ defines one adoptability assumption saying that variant GMV_i is a valid means to achieve the root goal if the required context $RCV_{i.j}$ holds. We exploit now simple statistical functions to define basic metrics that can be used by a system to select the best GMV based on the operation experience the system has:

– **Assumption Validity** (AV): this factor represents the statistical evidence concerning the capability of GMV_i to reach the root goal when a specific required context $RCV_{i.j}$ holds. Assumption validity uses the monitored data collected in Table 1a. Suppose GMV_i was enacted m times, out of which required context $RCV_{i.j}$ was true n times, out of which the GMV_i led to reach the root goal o times. AV is computed as the ration between the successful executions of GMV_i over all executions in which $RCV_{i.j}$ was true: $AV(GMV_i, RCV_{i.j}) = o/n$
– **Assumption Criticality** (AC): this metric represents the extent to which the falsity of a required context variant $RCV_{i.j}$ prevents GMV_i from achieving the root goal (though some other $RCV_{i.k}$ holds). Suppose GMV_i was enacted p times, out of which $RCV_{i.j}$ was false q times, out of which the $GMV_{i.j}$ did not lead to reach the root goal r times, then $AC(GMV_i, RCV_{i.j}) = r/q$

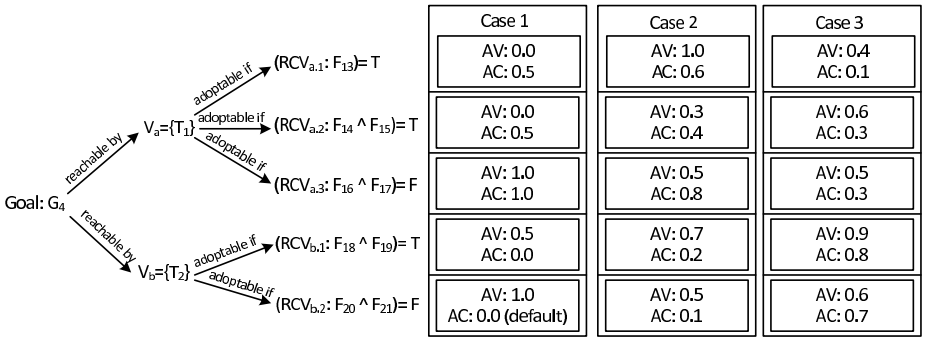


Fig. 3. Three different autonomic evolution scenarios involving the computation of AV and AC.

Fig. 3 shows that currently, $RCV_{a.1}$, $RCV_{a.2}$, and $RCV_{b.1}$ hold, whereas $RCV_{a.3}$ and $RCV_{b.2}$ do not hold. It also shows three different cases for the values of AV and AC. The first (Case 1) is computed on the basis of the operation history shown in Table 1a, whereas the other two reflect other operation histories. Upon that, the system will select the goal model variant that is the most likely to reach the root goal. Such likelihood is determined by considering both AV and AC of each pair $\langle GMV_i, RCV_{i.j}\rangle$. Instead of giving one algorithm to elect the GMV to enact, we here outline several policies that the designer could adopt, and probably change over time, that guide the decision making algorithm.

– **Optimistic**: this policy selects a GMV to enact on the basis of on the holding $RCVs$ having the highest AV metric. This policy is optimistic because it ignores that the very same GMV_i might currently have a false $RCV_{i.j}$ with a high $AC(GMV_i, RCV_{i.j})$ factor. In other words, this policy gives more importance to the positive evidence from the holding $RCVs$ than the negative impact from not-holding ones on the satisfaction of goals. For example, in Case 1 of Fig. 3, the selected variant will be V_b, given that its required context variant $RCV_{b.1}$ is the holding context having maximum AV value (0.5).

– **Sceptical**: this policy selects the goal model variant based on the lowest AC metric. This policy is sceptical because, irrespective of the likelihood of success of a GMV, it will choose a variant that, in the given context, is less likely to fail. The policy gives more importance to the negative impact the false $RCVs$ have on a GMV that the positive evidence the true $RCVs$ give. For example, in Case 2 of Fig. 3, the selected variant will be V_b, because its required context variant $RCV_{b.2}$ is that, among not-holding ones, having lowest AC value (0.1).
– **Balanced**: the selection according to this policy considers both the AV and the AC metrics. It is often the case that, considering AV and AC alone, the selected variant would be different. This is true, for instance, in Case 3 of Fig. 3, where the optimistic policy would choose V_b (due to the high $AV(V_a, RCV_{b.1})$ value which is 0.9), while the sceptical policy would choose V_a (due to the low $AC(V_a, RCV_{a.3})$ value which is 0.3). The balanced policy gives different weights to the two metrics, for instance 50% each. So, the balanced view will choose V_a, due to the $AV(V_a, RCV_{a.2})$ value which is 0.6 and $AC(V_a, RCV_{a.3})$ value which is 0.3.

4.3 Designer-supported Evolution of Refinement Assumptions

Many types of evolutionary actions concerning requirements models cannot be taken autonomously by software. This happens when evolution requires to apply substantial changes in the model, changes that are more radical than updating the rank of software variants which we described in Sec. 4.2. We focus here on evolutionary actions a designer can carry out on the basis of the operation experience history the system has gathered at runtime (Table 1b). We outline two primitive types of evolutionary actions that apply to requirements refinement (decomposition and specialization) assumptions, and explain when these actions should be applied. We illustrate them with the aid of the examples in Table 2.

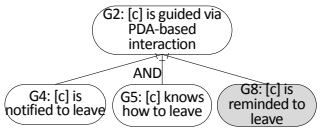
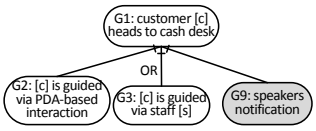
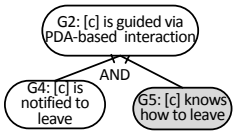
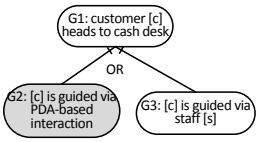
<table>
<tr>
<td>Add to decomposition. G_8 is added when operations like J_3 (Table 1b) occur often.
 
</td>
<td>Add to specialization. G_9 is added when operations like J_4 and J_5 occur often.
 
</td>
</tr>
<tr>
<td>Remove from decomposition. G_5 is removed when operations like J_7 rarely occur, while operations like J_6 occur often.
 
</td>
<td>Remove from specialization. G_2 is removed when operations like J_1 rarely occur, while operations like J_4 occur often.
 
</td>
</tr>
</table>

Table 2. Designer-supported evolution illustrated on the refinement of goal G_2 in Fig. 2.

Add sub-requirements: the refinement of a requirement is changed by adding a new sub-requirement. In a contextual goal model, a new sub-goal is added either to an AND-decomposition or to an OR-decomposition. The first case means that an additional sub-goal should be achieved to reach the parent goal. The second case corresponds to adding a new option to achieve the parent goal:

- Additions to AND-decompositions are needed when the operation history shows that, in many software operations, the achievement of the subgoals was not enough to achieve the parent goal. For example, operation J_3 means that a customer was successfully notified (G_4 was reached), he was informed about the way to leave (G_5 was reached), but still he did not leave on time (G_2 and G_1 were not reached). The system is supposed to continuously analyse the operation history (Table 1b) and, if operations like J_3 occur often, then it will ask the designer to take an addition evolutionary action. The designer could add a subgoal like "customer is reminded to leave" (G_8).
- Additions to OR-decompositions are required if the operation history shows that the achievement of alternative sub-goals, even after autonomous evolution, is typically not sufficient to reach the parent goal. For example, operation J_4 means that a customer was successfully guided by his PDA (he read the notification to leave and instructions about the way to leave, reaching therefore G_2), but he did not eventually leave on time (G_1 was not reached). J_5 represents a similar experience with respect to G_3. If operations like J_4 and J_5 occur often, then the system informs the designer asking him to add a new alternative. The designer could add a sub-goal such as "make announcement via the shopping mall public speakers" (G_9).

Remove sub-requirements: the refinement of a requirement is modified by removing a sub-requirement. In an AND-decomposition, a sub-goal is removed, meaning that to achieve the parent goal fewer sub-goals have to be achieved. In an OR-decomposition, removing a sub-goal means deleting an alternative way to achieve the parent goal.

- Removing a sub-goal from an AND-decomposition is applied when that sub-goal is typically unnecessary for the satisfaction of the parent goal. For example, in software operation J_6 the customer was successfully notified (G_4 was reached) but he did not read/receive instructions to leave (G_5 was not met), and still he moved towards the cash desk (G_2 and G_1 were reached). If the parent goal is often satisfied without G_5 being satisfied, the system will inform the designer suggesting to remove it. This will imply removing or disabling all software variants that support such goal from the implemented system.
- Removing a sub-goal from an OR-decomposition is applied when that sub-goal does not usually lead to the satisfaction of its parent goal. For example, if operations like J_4 (explained above) happen very often, this means that notifying customers and leading them by PDAs is an inapplicable alternative (for the root goal is not met) that need not be supported. Thus, the system will suggest the designer to remove this alternative and the corresponding software functionalities.

5 Conclusions and Future work

Requirements evolution is a main driver for software evolution. Requirements evolve due to many reasons. So far, literature focused mainly on changes in stakeholders' needs. Here, we advocated for and illustrated another main reason for requirements evolution; the assumptions in a requirements model. Assumptions validity is not predictable at design time and, moreover, changes over time. We conceive evolution as a lifelong process that moves software towards a behaviour based on the assumptions proven more valid. Evolution is desirably enacted by the system itself as an autonomic activity. However, certain kinds of evolution are not possible autonomously, and in this case the system can only announce problematic assumptions to designers, asking them to take an appropriate evolutionary action. To support these evolutions, software should monitor runtime operation and diagnose if the assumptions hold. We illustrated our view using contextual goal models as requirements models.

Future work involves mainly three threads. First, we will develop and implement algorithms that enact the principles we introduced in this paper and enable contextual requirements evolution. A crucial role will be played by the decision-making algorithm to evolve the rank of goal model variants. This should be a multi-factor algorithm that considers different dimensions such as operation history, qualities, preferences, timeliness, etc. Second, we will devise and investigate principles to adopt, compose, and switch between policies to select variants. Third, we will define automated reasoning techniques to identify the evolutionary actions suggested to designers and to select which is the best set of evolutionary actions that maximizes positive impact and minimizes costs.

Acknowledgments This work has been partially funded by the EU Commission, through the ANIKETOS, FastFix, SecureChange, and NESSOS projects and by Science Foundation Ireland grant $03/CE2/I303\_1$.

References

1. Lam, W., Loomes, M.: Requirements evolution in the midst of environmental change: a managed approach. In: Proceedings of CSMR 98. (1998) 121 –127
2. Harker, S., Eason, K., Dobson, J.: The change and evolution of requirements as a challenge to the practice of software engineering. In: Proceedings of RE '03. (1993) 266 –272
3. Finkelstein, A., Savigni, A.: A framework for requirements engineering for context-aware services. In: Proceedings of STRAW'01. (2001)
4. Salifu, M., Yu, Y., Nuseibeh, B.: Specifying monitoring and switching problems in context. In: Proceedings of RE '07. (2007) 211–220
5. Ali, R., Dalpiaz, F., Giorgini., P.: A goal modeling framework for self-contextualizable software. In: Proceedings of EMMSAD'09. (2009) 326–338
6. Hartmann, H., Trew, T.: Using feature diagrams with context variability to model multiple product lines for software supply chains. In: Proceedings of SPLC '08. (2008) 12–21
7. Ali, R., Dalpiaz, F., Giorgini, P.: A goal-based framework for contextual requirements modeling and analysis. Requirements Engineering **15** (2010) 439–458
8. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An agent-oriented software development methodology. Autonomous Agents and Multi-Agent Systems **8**(3) (2004) 203–236
9. Yu, E.: Modelling strategic relationships for process reengineering. Ph.D. Thesis, University of Toronto (1995)