Dynamic Requirements Specification for Adaptable and Open Service-Oriented Systems

Ivan J. Jureta, Stéphane Faulkner, Philippe Thiran

Information Management Research Unit, University of Namur, Belgium
iju@info.fundp.ac.be; sfaulkne@fundp.ac.be; pthiran@fundp.ac.be

Abstract. It is not feasible to engineer requirements for adaptable and open service-oriented systems (AOSS) by specifying stakeholders' expectations in detail during system development. Openness and adaptability allow new services to appear at runtime so that ways in, and degrees to which the initial functional and nonfunctional requirements will be satisfied may vary at runtime. To remain relevant after deployment, the initial requirements specification ought to be continually updated to reflect such variation. Depending on the frequency of updates, this paper separates the requirements engineering (RE) of AOSS onto the RE for: individual services (Service RE), service coordination mechanisms (Coordination RE), and quality parameters and constraints guiding service composition (Client RE). To assist existing RE methodologies in dealing with Client RE, the Dynamic Requirements Adaptation Method (DRAM) is proposed. DRAM updates a requirements specification at runtime to reflect change due to adaptability and openness.

1 Introduction

To specify requirements, the engineer describes the stimuli that the future system may encounter in its operating environment and defines the system's responses according to the stakeholders' expectations. The more potential stimuli she anticipates and accounts for, the less likely a discrepancy between the expected and observed behavior and quality of the system. Ensuring that the requirements specification is complete (e.g., [17]) becomes increasingly difficult as systems continue to gain in complexity and/or operate in changing conditions (e.g., [15, 10]). Adaptable and open service-oriented systems (AOSS) are one relevant response to such complexity. They are *open* to permit a large pool of distinct and competing services orignating from various service providers to participate. AOSS are *adaptable*—i.e., an AOSS coordinates service provision by dynamically selecting the participating services according to multiple quality criteria, so that the users continually receive optimal results (e.g., [7, 8]).

A complete requirements specification for an AOSS would include the description of all relevant properties of the system's operating environment, and of all alternative system and environment behaviors. All services that may participate would thus be entirely known at development time. Following any established

RE methodology (e.g., KAOS [4], Tropos [3]), such a specification would be constructed by moving from abstract stakeholder expectations towards a detailed specification of the entire system's behavior. As we explain in Section 2, applying such an approach and arriving at the extensive specification of an AOSS is not feasible. In response, this paper introduces concepts and techniques needed to *(1) determine how extensive the initial specification ought to be and what parts thereof are to be updated at runtime to reflect system adaptation*, and *(2) know how to perform such updates*. The specification can then be used to continually survey and validate system behavior. To enable (1), this paper separates the requirements engineering (RE) of AOSS depending on the frequency at which the requirements are to be updated (§2): RE executed for individual services or small sets of services (*Service RE*), RE of mechanisms for coordinating the interaction between services (*Coordination RE*), and RE of parameters guiding the runtime operation of the coordination mechanisms (*Client RE*). To address (2), this paper focuses on Client RE and introduces a method, called Dynamic Requirements Adaptation Method (DRAM) for performing Client RE for AOSS (§3). We close the paper with a discussion of related work (§4), and conclusions and indications for future effort (§5).

Motivation. The proposal outlined in the remainder resulted from the difficulties encountered in engineering requirements for an experimental AOSS, call it TravelWeb, which allows users to search for and book flights, trains, hotels, rental cars, or any combination thereof. Services which perform search and booking originate from the various service providers that either represent the various airlines and other companies, so that TravelWeb aggregates and provides an interface to the user when moving through the offerings of the various providers. Each provider can decide what options to offer to the user: e.g., in addition to the basics, such as booking a seat on an airplane, some airlines may ask for seating, entertainment, and food preferences, while others may further personalize the offering through additional options. We have studied elsewhere [7, 8] the appropriate architecture and service composition algorithms for TravelWeb. Here, we focus on the engineering of requirements for such systems.

2 Service, Coordination, and Client RE

To engineer the requirements for TravelWeb, a common RE methodology such as Tropos [3] would start with early and late requirements analyses to better understand the organizational setting, where dependencies between the service providers, TravelWeb, and end users would be identified, along with the goals, resources, and tasks of these various parties. Architectural design would ensue to define the sub-systems and their interconnections in terms of data, control, and other dependencies. Finally, detailed design would result in an extensive behavioral specification of all system components. While other methodologies, such as KAOS [4] involve a somewhat different approach, all move from high-level requirements into detailed behavioral specifications. The discussion below, however, concludes that such an approach is not satisfactory, because:

1) TravelWeb is open. Various hotels/airlines/rental companies may wish to offer or retract their services. Characteristics of services that may participate in TravelWeb at runtime is thus unknown at TravelWeb development time. Individual services are likely to be developed outside the TravelWeb development team, before or during the operation of TravelWeb. It is thus impossible to proceed as described for the entire TravelWeb—instead, it is more realistic to apply an established RE methodology locally for each individual service, and separately for the entire TravelWeb system, taking individual services as black boxes of functionality (i.e., not knowing their internal architecture, detailed design, etc.).

2) Resources are distributed and the system adapts. All services may or may not be available at all times. Moreover, individual services are often not sufficient for satisfying user requests—that is, several services from distinct providers may need to interact to provide the user with appropriate feedback. Adaptability in the case of TravelWeb amounts to changing service compositions according to service availability, a set of quality parameters, and constraints on service inputs and outputs (see, [8] for details). RE specific to the coordination of services carries distinct concerns from the RE of individual services.

3) Quality parameters vary. Quality (i.e., nonfunctional) parameters are used by the service composer as criteria for comparing alternative services and service compositions. Quality parameters are not all known at TravelWeb development time, for different services can be advertised with different sets of quality parameters. As the sets of quality parameters to account for in composing services change, (a) different sets of stakeholders' nonfunctional expectations will be concerned by various service compositions *and* (b) there may be quality parameters that do not have corresponding expectations in the initial specification. Observation (a) entails that initial desired levels of expectations may not be achieved at all times, making the initial specification idealistic. Deidealizing requirements has been dealt through a probabilistic approach by Letier and van Lamsweerde [11] where requirements are combined with probability of satisfaction estimates. In an adaptable system, the probability values are expected to change favorably over time (see, e.g., our experiments on service composition algorithms for AOSS [7, 8]), so that updating the initial requirements specification to reflect the changes seems appropriate if the specification is to remain relevant after system deployment. Observation (b) relates to the difficulty in translating stakeholders' goals into a specification: as March observed in a noted paper [12], both individual and organizational goals (which translate into requirements) tend to suffer from problems of relevance, priority, clarity, coherence, and stability over time, all of which relate to the variability, inconsistency, and imprecision, among other, of stakeholder preferences. Instead of assuming that the initial set of expectations is complete, the specification can be updated at runtime to reflect new system behaviors *and* to enable the stakeholders to modify requirements as they learn about the system's abilities and about their own expectations.

Having established that updates are needed, we turn to the question of what to update. A requirements specification for an AOSS involves requirements that are of different variability over time. Our experience with AOSS [7, 8] indicates

that a particular combination of service-oriented architecture and service coordination algorithm enables adaptability, whereby the architecture and the algorithm act as a cadre in which various requirements can be specified. Since adaptability does not require change in the architecture and algorithm, requirements on these two remain reasonably stable. This observation, along with the localization of service-specific RE to each individual service or small service groups leads to a separation of AOSS RE effort as follows:

Service RE involves the engineering of requirements for an individual service, or a set of strongly related services (e.g., those obtained by modularization of a complex service). Depending on whether the service itself is adaptable, a classical RE methodology such as Tropos or KAOS can be applied. As the coordination mechanism selects individual services for fulfiling user requests, requirements on an individual service do not change with changes in service requests (inputs and/or outputs and constraints on these and quality parameters change with variation in requests).

Coordination RE takes services as self-contained functionality and focuses on the requirements for the coordination of services. In an AOSS, this typically involves the definition of the architecture to enable openness, service interaction, service selection, and service composition for providing more elaborate, composite services to fulfil user requests. As noted above, these requirements vary less frequently than those elicited as a result of Client RE.

Client RE assumes a coordination mechanism is defined and is guided by constraints to obey, and quality parameters to optimize (e.g., QoS, execution time, service reputation). This is the case after a service-oriented architecture is defined in combination with an algorithm for service composition (see, e.g., [7, 8]). The aim at Client RE is to facilitate the specification of service requests at runtime. This involves, among other expressing constraints on desired outputs, quality criteria/parameters for evaluating the output and the way in which it is produced. This can be performed by traditional RE methodologies. In addition, Client RE ought to enable the definition of mechanisms for updating the service requests specification according to change in AOSS's behavior at runtime. The set of constraints and quality parameters is likely to vary as new services appear and other become unavailable. Quality parameter values will vary as well, as the system adapts to the availability of the various services and change in stakeholders' expectations.

3 Using DRAM at Client RE

We arrived above at the conclusion that there are two tasks to perform at Client RE: (a) specification of requirements that result in service requests, and (b) the definition of mechanisms for keeping these requirements current with behaviors of the AOSS and degrees of quality it can achieve over the various quality parameters defined in the requirements. We focus now on Client RE, assume the use of an established RE methodology for accomplishing (a), and introduce the Dynamic Requirements Adaptation Method (DRAM) to perform (b). DRAM is

thus not a standalone RE methodology—it does not indicate, e.g., how to elicit stakeholder expectations and convert these into precise requirements. Instead, DRAM integrates concepts and techniques for defining mappings between fragments of the requirements specification produced by an existing RE methodology and elements defining a service request (SReq). Mapping requirements onto SReqs aims to ensure that the stakeholders' expectations are translated into constraints and quality parameters understood by the AOSS. Mapping in the other direction—from SReqs onto requirements—allows the initial (also: static) requirements specification to be updated to reflect runtime changes in the system due to adaptability and openness. The specification obtained by applying DRAM on the initial, static requirements specification is referred to as the *dynamic requirements specification*.

Definition 1. ***Dynamic requirements specification*** $\mathcal{S}$ *is* $\langle \mathsf{R}, \mathcal{R}, \mathcal{Q}, \mathcal{P}, \mathcal{U}, \mathcal{A} \rangle$*, where:* R *is the* ***static requirements specification*** *(Def.2);* $\mathcal{R}$ *the set of* ***service requirements*** *(Def.3);* $\mathcal{Q}$ *the set of* ***quality parameters*** *(Def.4);* $\mathcal{P}$ *the* ***preferences specification*** *(Def.5);* $\mathcal{U}$ *the set of* ***update rules*** *(Def.6); and* $\mathcal{A}$ *the* ***argument repository*** *(Def.7).*

The aim with DRAM is to build the dynamic requirements specification. Members of R are specifications of nonfunctional and functional requirements, taking the form of, e.g., goals, softgoals, tasks, resources, agents, dependencies, scenarios, or other, depending on the RE methodology being used. Service requests submitted at runtime express these requirements in a format understandable to service composers in the AOSS. Nonfunctional requirements from R are mapped onto elements of $\mathcal{Q}$ and $\mathcal{P}$, whereas functional requirements from R onto service request constraints grouped in $\mathcal{R}$. As equivalence between fragments of R and $\mathcal{R}, \mathcal{Q}, \mathcal{P}$ can seldom be claimed, a less demanding binary relation is introduced: the *justified correspondence* "$\triangleq$" between two elements in $\mathcal{S}$ indicates that there is a *justification* for believing that the two elements correspond in the given AOSS, at least until a defeating argument is found which breaks the justification. In other words, the justified correspondence establishes a mapping between instances of concepts and relationships in the language in which members of R are written and the language in which members of $\mathcal{R}, \mathcal{Q}, \mathcal{P}$ are written. The preferences specification $\mathcal{P}$ contains information needed to manage conflict and subsequent negotiation over quality parameters that cannot be satisfied simultaneously to desired levels. Update rules serve to continually change the contents of R according to system changes at runtime. Finally, the argument repository $\mathcal{A}$ contains knowledge, arguments, and justifications used to construct justified correspondences and at other places in $\mathcal{S}$, as explained below.

$\mathcal{S}$ is continually updated to reflect change in how the service requests are fulfilled. Updates are performed with update rules: an update rule will automatically (or with limited human involvement) change the R according to the quality parameters, their values, and the constraints on inputs and outputs characterizing the services composed at runtime to satisfy service requests. An update rule can thus be understood as a mapping between fragments of R and those of

$\mathcal{R}, \mathcal{Q}, \mathcal{P}$. Consequently, an update rule is derived from a justified correspondence. It is according to the constraints on inputs/outputs and quality parameter values observed at runtime that fragments of requirements will be added or removed to R. Update rules work both ways, i.e., change in R is mapped onto service requests, and the properties of services participating in compositions are mapped onto fragments of R.

Building fully automatic update rules is difficult for it depends on the precision of the syntax and semantics of languages used at both ends, i.e., the specification language of the RE methodology which produces R and the specification language employed to specify input/output constraints on services and quality parameters. Due to a lack of agreement on precise conceptualizations of key RE concepts (e.g., [17]), DRAM makes no assumptions about the languages employed for writing R, $\mathcal{R}$, and $\mathcal{Q}$. Hence the assumption that languages at both ends are ill-defined, and the subsequent choice of establishing a "justified" correspondence (i.e., a defeasible relation) between specification fragments. An unfortunate consequence is that update rules in many cases cannot be established automatically—a repository of update rules is built during testing and at runtime. $\mathcal{S}$ integrates the necessary means for constructing update rules: to build justified correspondences between elements of R and $\mathcal{R}, \mathcal{Q}, \mathcal{P}$, arguments are built and placed in the argument repository $\mathcal{A}$. Update rules are automatically extracted from justified correspondences. As competing services will offer different sets of and values of quality parameters at service delivery, and as not all will be always available, trade-offs performed by the AOSS need to be appropriately mapped to R. Moreover, stakeholders may need to negotiate the quality parameters and their values. $\mathcal{P}$ performs the latter two roles. DRAM proceeds as follows in building the dynamic requirements specification (concepts and techniques referred to below are explained in the remainder).

Building the dynamic requirements specification with DRAM

1. Starting from the static requirements specification R (Def.2), select a fragment r $\in$ R of that specification that has not been converted into a fragment in $\mathcal{R}$ (Def.3), $\mathcal{Q}$ (Def.4), and/or $\mathcal{P}$ (Def.5).
2. Determine the service requirement and/or quality parameter information that can be extracted from r as follows:
   (a) If r is a functional requirement (i.e., it specifies a behavior to perform), focus is on building a justified correspondence (see, Def.6 and Technique 1) between r and elements of service requirements. Consider, e.g., the following requirement: Each user of TravelWeb expects a list of available flights for a destination to be shown within 5 seconds after submitting the departure and destination city and travel dates.
   $available(depC, depD, arrC, arrD, flight) \wedge correctFormat(depC, depD, arrC, arrD) \Rightarrow \diamond_{5s} shown(searchResults, flight)$
   Starting from the above functional requirement:
      i. Identify the various pieces of data that are to be used (in the example: $depC, depD, arrC, arrD, flight$) and those that are to be produced ($searchResults$) according to the requirement.
      ii. Find services that take the used data as input and give produced data at output (e.g., FlightSearch Serv, s.t. $\{depC, depD, arrC, arrD, flight\} \subseteq I \wedge searchResults \in O$).

iii. Determine whether the service requirements available on inputs justifiably corresponds to the conditions on input data in the requirement, and perform the same for output data (i.e., check if there is a justified correspondence between input/output service requirements and conditions in the relevant requirements in R—i.e., use Def.6 and Technique 1). If constraints do not correspond (justified correspondence does not apply), map the conditions from the requirement in R into constraints on inputs and/or outputs, and write them down as service requirements. If there is no single service that satisfies the requirement (i.e., step 2(a)i above fails), refine the requirement (i.e., brake it down into and replace with more detailed requirements)—to refine, apply techniques provided in the RE methodology.

iv. Use step 2b to identify the quality parameters and preferences related to the obtained service requirement.

(b) If r is a nonfunctional requirement (i.e., describes *how* some behavior is to be performed, e.g., by optimizing a criterion such as delay, security, safety, and so on), the following approach is useful:

i. Find quality parameters (Def.4) that describe the quality at which the inputs and outputs mentioned in a particular service requirement are being used and produced. In the example cited in the DRAM process, the delay between the moment input data is available and the moment it is displayed to the user can be associated to a quality parameter which measures the said time period.

ii. Following Def.4, identify the various descriptive elements for each quality parameter. Use R as a source for the name, target and threshold value, and relevant stakeholders. If, e.g., Tropos is employed to produce R, softgoals provide an indication for the definition of quality parameters.

iii. For each quality parameter that has been defined, specify priority and preferences. Initial preferences data for trade-offs comes from test runs.

3. Write down the obtained $r \in \mathcal{R}$, $q \in \mathcal{Q}$, and/or $p \in \mathcal{P}$ information, along with arguments and justifications used in mapping r into r and/or q. Each justified correspondence obtained by performing the step 2. above is written down as an update rule $u \in \mathcal{U}$.
4. Verify that the new arguments added to $\mathcal{A}$ do not defeat justifications already in $\mathcal{A}$; revise the old justifications if needed.

Definition 2. *The* ***static requirements specification*** R *is the high-level requirements specification obtained during RE before the system is in operation.*

R is obtained by applying a RE methodology, such as, e.g., KAOS [4] or Tropos [3]. The meaning of "high-level" in Def.2 varies accross RE methodologies: if a goal-oriented RE methodology is employed, R must contain the goals of the system down to the operational level, so that detailed behavioral specification in terms of, e.g., state machines, is not needed. If, e.g., KAOS is used, the engineer need not move further than the specification of goals and concerned objects, that is, can stop before operationalizing goals into constraints. If Tropos is used, the engineer stops before architectural design, having performed late requirements analysis and, ideally, formal specification of the functional goals.

Example 1. When a RE methodology with a specification language grounded temporal first-order logic is used[1], the following requirement $\mathsf{r} \in \mathsf{R}$ for TravelWeb states that all options that a service may be offering to the user should be visible to the first time user:

$\mathsf{1stOpt} \equiv (hasOptions(servID) \wedge firstTimeUser(servID, userID)$
$\Rightarrow \diamond_{1s} showOptions(all, servID, userID))$

Definition 3. *A **service requirement** $r \in \mathcal{R}$ is a constraint on service inputs or outputs that appears in at least one service request and there is a unique $\mathsf{r} \in \mathsf{R}$ such that there is a justified correspondence between it and r.*

Example 2. Any service that visualizes to the TravelWeb user the options that other services offer when booking obeys the following service requirement:

$r =$ (**input:**servID $\notin$ userID.visited $\wedge$ servID.options $\neq \emptyset$; **output:**thisService.show $=$ servID.options)

Definition 4. *A **quality parameter** $q \in \mathcal{Q}$ is a metric expressing constraints on how the system (is expected to) performs. $q = \langle$Name, Type, Target, Threshold, Current, Stakeholder$\rangle$, where Name is the unique name for the metric; Type indicates the type of the metric; Target gives a unique or a set of desired values for the variable; Threshold carries the worst acceptable values; Current contains the current value or average value over some period of system operation; and Stakeholder carries names of the stakeholders that agree on the various values given for the variable.*

Example 3. The following quality parameters can be defined on the service from Example 2:

$q_1 = \langle$ShowDelay, Ratio, 500ms, 1s, 780ms, MaintenanceTeam$\rangle$
$q_2 = \langle$OptionsPerScreen, Ratio, {3,4,5}, 7, (all), UsabilityTeam$\rangle$
$q_3 = \langle$OptionsSafety, Nominal, High, Med, Low, MaintenanceTeam$\rangle$
$q_4 = \langle$BlockedOptions, Ratio, 0, ($\geq$ 1), 0, MaintenanceTeam$\rangle$

As quality parameters usually cannot be satisfied to the ideal extent simultaneously, the preference specification contains information on priority and positive or negative interaction relationships between quality parameters. Prioritization assists when negotiating trade-offs, while interactions indicate trade-off directions between parameters.

Definition 5. *The **preferences specification** is the tuple $\mathcal{P} = \langle \succ, \mathcal{P}^{\succ}, \mathcal{P}^{\pm} \rangle$. "$\succ$" is a priority relation over quality parameters. The set $\mathcal{P}^{\succ}$ contains partial priority orderings, specified as $(q_i \succ q_j,$ Stakeholder$) \in \mathcal{P}^{\succ}$ where q_i carries higher priority than q_j, and Stakeholder contains the names of the stakeholders agreeing on the given preference relation. Higher priority indicates that a trade-off between*

[1] Assuming, for simplicity, a linear discrete time structure, one evaluates the formula for a given history (i.e., sequence of global system states) and at a certain time point. The usual operators are used: for a history H and time points i, j, $(H, i) \models \circ\phi$ iff $(H, next(i)) \models \phi$; $(H, i) \models \diamond\phi$ iff $\exists j > i, (H, j) \models \phi$; $(H, i) \models \Box\phi$ iff $\forall j \geq i, (H, j) \models \phi$. Mirror operators for the past can be added in a straightforward manner. Operators for eventually $\diamond$ and always $\Box$ can be decorated with duration constraints, e.g., $\diamond_{\leq 5s}\phi$ indicates that ϕ is to hold some time in the future but not after 5 seconds. To avoid confusion, note that $\rightarrow$ stands for implication, while $\phi \Rightarrow \psi$ is equivalent to $\Box(\phi \rightarrow \psi)$. For further details, see, e.g., [16].

the two quality parameters will favor the parameter with higher priority. The set $\mathcal{P}^{\pm}$ *contains interactions. An interaction indicates that a given variation of the value of a quality parameter results in a variation of the value of another quality parameter. An interaction is denoted* $(q_1 \overset{b_1 \Rightarrow b_2}{\longleftrightarrow} q_2)@\phi$. $q_1 \overset{b_1 \Rightarrow b_2}{\longleftrightarrow} q_2$ *indicates that changing the value of the quality parameter* q_1 *by or to* b_1 *necessarily leads the value of the parameter* q_2 *to change for or to* b_2. *As the interaction may only apply when particular conditions hold, an optional non-empty condition* ϕ *can be added to indicate when the interaction applies. The condition is written in the same language as service requirements. When the relationship between the values of two quality parameters can be described with a function, we give that functional relationship instead of* $b_1 \Rightarrow b_2$.

Example 4. Starting from the quality parameters in Ex.3, the following is a fragment of the preferences specification:

$$p_1 = \left(\mathsf{OptionsPerScreen} \overset{+1 \Rightarrow +60ms}{\longleftrightarrow} \mathsf{ShowDelay}\right) @(\mathsf{OptionsPerScreen} > 4)$$

p_1 indicates that increasing the number of options per screen by 1 increases the delay to show options to the user by 60ms, this only if the number of options to show is above 4.

Definition 6. *A* ***justified correspondence*** *exists between* $\phi \in \mathsf{R}$ *and* $\psi \in \mathcal{R} \cup \mathcal{Q} \cup \mathcal{P}$, *i.e.,* $\phi \triangleq \psi$ *iff there is a justification* $\langle P, \phi \triangleq \psi \rangle$.

Recall from the above that the justified correspondence is a form of mapping in which very few assumptions are made on the precision and formality of the languages being mapped. This entails the usual difficulties (as those encountered in ontology mapping, see, e.g., [9]) regarding conversion automation and the defeasibility of the constructed mappings, making DRAM somewhat elaborate to apply in its current form. Defeasibility does, however, carry the benefit of flexibility in building and revising mappings.

Definition 7. *A* ***justification*** $\langle P, c \rangle$ *is an argument that remains undefeated after the justification process.*[2]

[2] Some background [14]: Let A a set of agents (e.g., stakeholders) and the first-order language $\mathcal{L}$ defined as usual. Each agent $a \in A$ is associated to a set of first-order formulae K_a which represent knowledge taken at face value about the universe of discourse, and Δ_a which contains defeasible rules to represent knowledge which can be revised. Let $K \equiv \bigcup_{a \in A} K_a$, and $\Delta \equiv \bigcup_{a \in A} \Delta_a$. "$\mid\!\sim$" is called the *defeasible consequence* and is defined as follows. Define $\Phi = \{\phi_1, \ldots, \phi n\}$ such that for any $\phi_i \in \Phi$, $\phi_i \in K \cup \Delta^{\downarrow}$. A formula ϕ is a defeasible consequence of Φ (i.e., $\Phi \mid\!\sim \phi$) if and only if there exists a sequence $B_1, \ldots, B_m$ such that $\phi = B_m$, and, for each $B_i \in \{B_1, \ldots, B_m\}$, either B_i is an axiom of $\mathcal{L}$, or B_i is in Φ, or B_i is a direct consequence of the preceding members of the sequence using modus ponens or instantiation of a universally quantified sentence. An argument $\langle P, c \rangle$ is a set of consistent premises P supporting a conclusion c. The language in which the premises and the conclusion are written is enriched with the binary relation $\hookrightarrow$. The relation $\hookrightarrow$ between formulae α and β is understood to express that "reasons to believe in the antecedent α provide reasons to believe in the consequent β". In short, $\alpha \hookrightarrow \beta$ reads "α is reason for β" (see, [14] for details). Formally then, P is an argument for c, denoted $\langle P, c \rangle$, iff: (1) $K \cup P \mid\!\sim c$ (K and P derive c); (2) $K \cup P \nvdash \bot$ (K and P are consistent); and (3) $\nexists P' \subset P, K \cup P' \mid\!\sim c$ (P is minimal for K).

Up to this point, the concepts needed in DRAM have been introduced. The remainder of this section describes the techniques in DRAM that use the given concepts in the aim of constructing the dynamic requirements specification.

Technique 1. The *justification process* [14] consists of recursively defining and labeling a *dialectical tree* $\mathcal{T}\langle P, c\rangle$ as follows:
1. A single node containing the argument $\langle P, c\rangle$ with no defeaters is by itself a dialectical tree for $\langle P, c\rangle$. This node is also the root of the tree.
2. Suppose that $\langle P_1, c_1\rangle, \ldots, \langle P_n, c_n\rangle$ each defeats[3] $\langle P, c\rangle$. Then the dialectical tree $\mathcal{T}\langle P, c\rangle$ for $\langle P, c\rangle$ is built by placing $\langle P, c\rangle$ at the root of the tree and by making this node the parent node of roots of dialectical trees rooted respectively in $\langle P_1, c_1\rangle, \ldots, \langle P_n, c_n\rangle$.
3. When the tree has been constructed to a satisfactory extent by recursive application of steps 1) and 2) above, label the leaves of the tree *undefeated* (U). For any inner node, label it undefeated if and only if every child of that node is a *defeated* (D) node. An inner node will be a defeated node if and only if it has at least one U node as a child. Do step 4 below after the entire dialectical tree is labeled.
4. $\langle P, c\rangle$ is a *justification* (or, P justifies c) iff the node $\langle P, c\rangle$ is labelled U.

Example 5. Fig.1 contains the dialectical tree for the justified correspondence $\mathsf{1stOpt} \triangleq r$, where r is from Ex.1 and r from Ex.2. To simplify the presentation of the example, we have used both formal and natural language in arguing. More importantly, notice that the correspondence $\mathsf{1stOpt} \triangleq r$ is unjustifed, as it is defeated by an undefeated argument containing information on a quality parameter and a fragment of the preferences specification. A justified correspondence such as, e.g., $\mathit{firstTimeUser}(\mathit{servID}, \mathit{userID}) \triangleq \mathit{servID} \notin \mathit{userID.visited}$, becomes an update rule, i.e., $(\mathit{firstTimeUser}(\mathit{servID}, \mathit{userID}) \triangleq \mathit{servID} \notin \mathit{userID.visited}) \in \mathcal{U}$. Having established that justified correspondence, the service requirement is taken to correspond to the given initial requirement until the justified correspondence is defeated. Elements of the argument repository correspond to the argument structure shown in Fig.1.

4 Related Work

Engineering requirements and subsequently addressing completeness concerns for AOSS has only recently started to receive attention in RE research. Berry and colleagues [1] argue in a note that, while much effort is being placed in enabling adaptive behavior, few have dealt with how to ensure correctness of software before, during, and after adaptation, that is, at the RE level. They recognize that RE for such systems is not limited to the initial steps of the system development process, but is likely to continue in some form over the entire

[3] Roughly (for a precise definition, see [14]) the argument $\langle P_1, c_1\rangle$ *defeats at* c an argument $\langle P_2, c_2\rangle$ if the conclusion of a subargument $\langle P, c\rangle$ of $\langle P_2, c_2\rangle$ contradicts $\langle P_1, c_1\rangle$ and $\langle P_1, c_1\rangle$ is more specific (roughly, contains more information) than the subargument of $\langle P_2, c_2\rangle$.

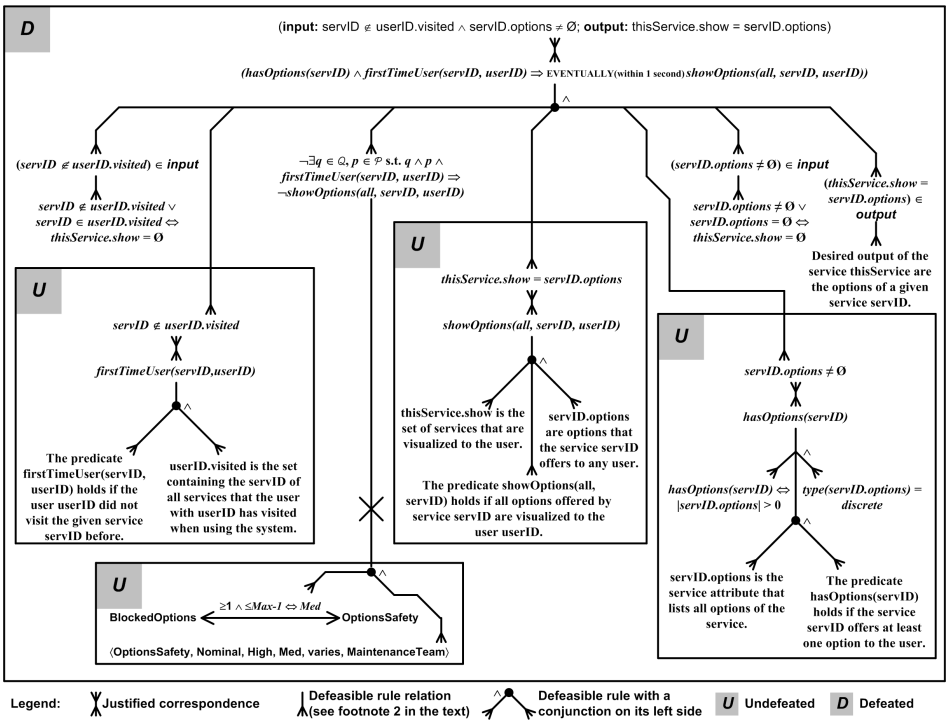

Fig. 1. Output of the justification process related to Examples 1 and 2.

lifecycle of the system. Zhang and Cheng [19] suggest a model-driven process for adaptive software; they represent programs as state machines and define adaptive behaviors usually encountered in adaptable systems as transitions between distinct state machines, each giving a different behavior to the system. Being situated more closely to the design phase of development than to RE, Zhang and Cheng's process has been related [2] to the KAOS RE methodology by using A-LTL instead of temporal logic employed usually in KAOS. In the extended KAOS, a requirement on adaptation behavior amounts to a goal refined into two sequentially ordered goals, whereby the first in the sequence specifies the conditions holding in the state of the system before adaptation while the second goal gives those to hold in the state after adaptation. This paper differs in terms of concerns being addressed and the response thereto. The suggested separation onto Service, Coordination, and Client RE for AOSS usefully delimits the concerns and focus when dealing with AOSS. The notion of dynamic requirements specification, along with the associated concepts and techniques is novel with regards to the cited research.

5 Conclusions and Future Work

This paper presents one approach to addressing the difficulties in the RE of AOSS. We argued that the RE of AOSS involves the specification of requirements that may vary at runtime. We consequently identified the most variable class of AOSS requirements and proposed DRAM, a method for specifying these within dynamic requirements specifications. The method has the benefit that it can be combined to any RE methodology. Its principal limitation at this time is the lack of automated means for defining or facilitating the definition of update rules. Automation of the DRAM process by reusing results in defeasible logic programming is the focus of current work.

References

1. D. M. Berry, B. H. Cheng, J. Zhang. The four levels of requirements engineering for and in dynamic adaptive systems. *REFSQ'05*.
2. G. Brown, B. H. C. Cheng, H. Goldsby, J. Zhang. Goal-oriented Specification of Adaptation Semantics in Adaptive Systems. *SEAMS@ICSE'06*.
3. J. Castro, M. Kolp, J. Mylopoulos. Towards requirements-driven information systems engineering: the Tropos project. *Info. Sys.*, 27(6), 2002.
4. A. Dardenne, A. van Lamsweerde, S. Fickas. Goal-directed requirements acquisition. *Sci. Comp. Progr.*, 20, 1993.
5. N. R. Jennings. On Agent-Based Software Engineering. *Artif. Int.*, 117, 2000.
6. I. J. Jureta, S. Faulkner, P.-Y. Schobbens. Justifying Goal Models. *RE'06*.
7. I. J. Jureta, S. Faulkner, Y. Achbany, M. Saerens. Dynamic Task Allocation within an Open Service-Oriented MAS Architecture. *AAMAS'07*. To appear.
8. I. J. Jureta, S. Faulkner, Y. Achbany, M. Searens. Dynamic Web Service Composition within a Service-Oriented Architecture. *ICWS'07*. To appear.
9. Y. Kalfoglou, M. Schorlemmer. Ontology Mapping: The State of the Art. *Dagstuhl Seminar Proceedings*, 2005.
10. J. O. Kephart, D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–52, 2003.
11. E. Letier, A. van Lamsweerde. Reasoning about partial goal satisfaction for requirements and design engineering. *ACM Sigsoft Softw. Eng. Notes* 29(6), 2004.
12. J. March. Bounded Rationality, Ambiguity, and the Engineering of Choice. *The Bell J. Econonomics*, 9(2), 1978.
13. M. P. Papazoglou, D. Georgakopoulos. Service-Oriented Computing. *Comm. ACM*, 46(10), 2003.
14. G. R. Simari, R. P. Loui. A mathematical treatment of defeasible reasoning and its implementation, *Artif. Int.*, 53, 1992.
15. D. Tennenhouse. Proactive Computing. *Comm. ACM*, 42(5), 2000.
16. A. van Lamsweerde, E. Letier. Handling Obstacles in Goal-Oriented Requirements Engineering. *IEEE Trans. Softw. Eng.*, 26(10), 2000.
17. A. van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. *RE'01*.
18. J. Zhang, B. H. C. Cheng. Specifying adaptation semantics. *WADS'05*.
19. J. Zhang, B. H. C. Cheng. Model-Based Development of Dynamically Adaptive Software. *ICSE'06*.